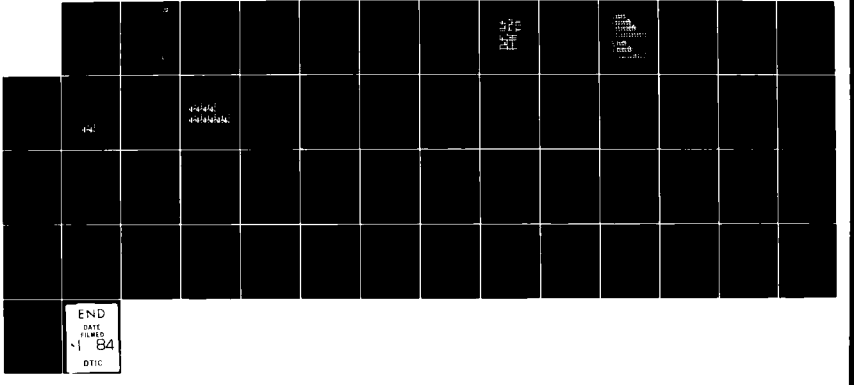
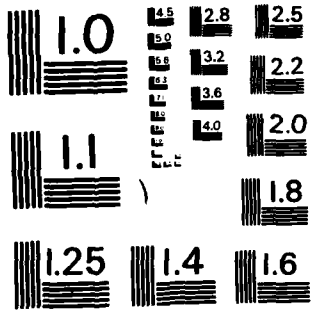
AD-A136 551 PRINCETON VLSI PROJECT(U) PRINCETON UNIV NJ DEPT OF 1/1
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE R J LIPTON
1982 N00014-82-K-0549

UNCLASSIFIED F/G 9/5 NL



END
DATE FILMED
1 84
DTIC



Princeton VLSI Project: Semi-Annual Report

Period Ending: November 1, 1982

Richard J. Lipton
Principal Investigator

EECS Department
Princeton University

Faculty:
B. W. Arden, D. Dobkin, H. Garcia-Molina, A. LaPaugh, K. Steiglitz, J. Valdes

PhD. Students:
D. Field, S. North, V. Ramachandran, G. Vijayan, A. Widgerson

Contract N00014-82-K-0549

Princeton VLSI Project: Semi-Annual Report

R. J. Lipton

1. Introduction

We have been officially underway at Princeton since August; hence, this report actually covers about three months of activity. There are three major aspects to our project: ALI, Census, and Testing.

2. ALI

ALI, our procedural language for VLSI design and layout, is now up and running, [4]. It has already been used by Dobkin and Drysdale to redesign a divider they previously designed on a graphics system at Xerox. In addition, LaPaugh's VLSI class has already used ALI for their VLSI projects. Dobkin is currently beginning to explore ways to create graphics interfaces to the ALI system.

ALI2 the second version of our layout system is coming along well. Valdes has already fully defined the new language and implementation is now under way [7]. ALI2 differs from ALI in two essential ways. First, it is based on, what we believe to be a much cleaner set of primitive and constructs. For instance, it does not have the "shuffle property" so many design languages do: by this we mean that the *order* of placement commands does matter. Second, ALI2 generates far fewer constraints than ALI does. This is of course critical if such constraint based systems are to be able to handle large complex layouts.

Plans are already underway on how to best exploit the unique features of ALI2. Ramachandran has just finished a study of the cost of increasing the size of drivers in order to speed up circuits [5]. She assumes that sizes of transistors can be changed but that the layout cannot be restructured. Under these assumptions she can tightly bound the worst case cost in terms of area of making the delays on all wires a "constant". Since ALI2 allows a designer to easily change the sizes of wires and transistors we feel that such results are important. Already it is common place for designers to size transistors for speedup in an *ad hoc* way [6].

Also Vijayan and Wigderson have isolated a number of new layout problems that arise naturally when one considers the implementation of ALI2 [8,9]. All these problems concern the embedding of "rectilinear" graphs. These are graphs where each edge is connected to the "left", "top", "right",or "bottom" of each vertex. They are currently exploring the computational complexity of a variety of layout problems here. For example, they can quickly recognize those graphs that have planar embeddings; moreover, they can also quickly find such embeddings when they exist.

3. Census

The census language is a new way to express parallel algorithms that use a fairly loosely coupled method of control, [3]. Work is under way to understand the limits and powers of such languages. North is beginning to identify those problem areas that can be successfully mapped onto the census language. We are also thinking about implementations of census, but no implementation is yet started.

Census has been looked at from a theory point of view by Chandra, Fortune,

and Lipton [2]. They have been able to get tight upper and lower bounds on the size of boolean circuits with *unbounded* fan-in. Unbounded fan-in circuits not only model an important class of census computations but they also model circuits such as PLA's. For example, it is possible to construct a circuit that adds two n-bit numbers in *constant* time and whose size is approximately linear in n. We are beginning to examine the feasibility of using these ideas in real VLSI designs.

4. Testing

Our work on testing divides into two areas. Arden is beginning to work on *first silicon testing* especially with respect to scanning electron microscopes (SEM). He plans to be on leave this spring and work with the SEM group in Munich. In the future we expect that we will be able to use a SEM that the Princeton Siemens group is about to get.

LaPaugh and Lipton have begun to work in the area of *production testing*. They have already successfully been able to completely characterize the testability of "prefix computations". Prefix computations arise naturally in a number of places: for instance, in carry-look-ahead adders. Such characterizations link the self-test of these computations with classic semi-group theory.

Also work is under way on a new self-test strategy which we call "toggle search". This method first generates the vector of all 0's; next it randomly changes one of the 0's to a 1; it repeats this until all 0's have been toggled to 1's, then the entire process begins again. Toggle search and its generalizations are well suited to many test environments where one bit of an input can be changed more quickly than a whole input vector. Already we have empirical evidence of the superiority of toggle search over other methods. Colleagues at IBM Watson Research have tested toggle on examples of about one thousand gates and found that it is about three times faster than standard methods. For example, it took about 600 thousand vectors versus 1.8 million to achieve 100% fault coverage on one piece of random control logic. We plan further experiments to further validate these results.

Finally, we have also found a way to transform any combinational logic circuit into one that is easy to test. Here by easy to test we mean that we can detect a very large class of physical faults. The penalty for this transformation is that the number of gates can increase by as much as 100%. We plan this coming year to carefully explore this new method. In particular, we wish to both understand the cost of the method and the class of faults it can and cannot detect.

5. References

[1] Brent, R.P., Kung, H.T., "The Chip Complexity of Binary Arithmetic," *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, ACM (New York), April, 1980.

[2] Chandra, A., Fortune, S., Lipton, R.J. - in preparation.

[3] Lipton, R.J., Valdes, J., "Census Functions: an Approach to VLSI Upper Bounds," *Proceedings 22nd IEEE Foundations of Computer Science* Nashville, Tennessee, October, 1981.

[4] Lipton, R.J., North, S.C., Sedgewick, R., Vijayan, G., "ALI: A Procedural Language to Describe VLSI Layouts," *Proceedings of the Nineteenth ACM-IEEE Design Automation*, Las Vegas, Nevada, June, 1982.

[5] Ramachandran, V., "On Driving Many Long Lins in a VLSI Layout," to appear in *Proceedings of the 23rd Foundations of Computer Science*, Chicago, Illinois, 1982.

[6] Shoji, M., "Electrical Design of BELLMAC-32A Microprocessor" in *Proceedings of ICCC-82*, New York, New York, 1982.

[7] Valdes, J., "ALI2 Implementation Notes," - unpublished manuscript.

[8] Vijayan, G., "Completeness of VLSI Layouts," VLSI Memo #1, Department of EECS, Princeton University.

[9] Vijayan, G., Wigderson, A., "Rectilinear Graphs and their Embeddings," - in preparation.

VLSI Layout as Programming

Richard J. Lipton†
Stephen C. North•
Robert Sedgewick‡
Jacobo Valdes†
Gopalakrishnan Vijayan†

†Department of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ

‡Computer Science Department
Brown University
Providence, RI

•Bell Laboratories
Murray Hill, NJ

Abstract: The first component of a VLSI design environment being built at Princeton is described. The general theme of this effort is to make the design of VLSI circuits as similar to programming as possible. We are trying to build tools that do for the VLSI circuit designer what the best software tools do for the implementor of large software systems. The work described here is a procedural language to specify circuit layouts.

1. Introduction

In this paper we describe a very important component of any VLSI design environment: a tool to automate the layout of circuits. This work is part of an effort to create an integrated environment for VLSI design (including layout systems, device and switch level simulators and testing facilities) currently under way at Princeton.

Our main thesis is that the VLSI design task can be profitably thought of as a *programming task*, as opposed to a geometric editing task. We believe that much is to be gained by consciously attempting to apply our knowledge about programming to this new activity. We have thus tried to create tools for the VLSI designer that incorporate the most useful features of the software develpoment tools that we are familiar with.

Although we feel that we have had moderate success in this endeavor we are well aware of how much room for improvement we have left, and would like to help convince the community of people interested in the design of programming language and programming environments that there are fresh and important challenges in this relatively new direction.

A prototype of the procedural layout language described in this paper has been operational for some months. All figures given in thic paper were generated by the language and all the code fragments have been used as part of larger programs.

Portions of this paper appeared in the Proceedings of the 1982 ACM symposium on Principles of Programming Languages and in the Proceedings of the 1982 Design Automatic Conference.

2. ALI: a procedural language to describe layouts

The main feature of ALI as a layout language is that it allows its user to design layouts at a *conceptual level* in which neither sizes nor positions (absolute or relative) of layout components may be specified. Mostly as a consequence of this, ALI simultaneously (i) makes the layout task more like programming than editing, (ii) eliminates the need for design rule checking after the layout is generated, (iii) permits the creation of easy to use cell libraries and (iv) provides the designer with the mechanisms to describe a layout hierarchically so that most of the detail at one level of the hierarchy is truly hidden from all higher levels.

The notion of not assigning sizes or positions to any object in a layout until the complete layout has been described (similar to the idea of *delayed binding* in programming languages), sets ALI apart not only from just about all of the graphics based layout editors we know of ([3], [4], [7], [13], [17]) but also – with the exception of [14] -- from most of the procedural languages for the layout task currently in use or recently proposed, whether or not they include a graphics interface ([1], [4], [5], [8], [9], [10], [15]).

The issues that we tried to address with ALI are the following.

- The creation of an *open ended tool*. Graphics editors tend to be closed tools in that it is hard to automate the layout process beyond what the original design of the system allowed. Procedural languages are generally much better in this respect. However, the fact that most such languages require the specification of absolute sizes and positions, makes the creation of a general purpose library of cells a hard task, since information about the sizes and positions of the cell elements that can interact with the outside world has to be apparent to the user of the library. The absence of absolute sizes and positions makes this problem much less severe in ALI. The extensibility of ALI derives from the fact that it has been built on top of Pascal, thereby making the full power of Pascal available to the designer. The generation of tools to automate the layout process, such as simple routers or PLA generators, involves writing Pascal routines to solve some abstract version of the problem and having done so invoke ALI cells to generate the layouts
- Creating tools that are *simple to use and easy to learn*. In particular, we want to avoid tools whose behavior is unpredictable. Many programs which rely heavily on sophisticated heuristics respond to small changes in their input with wholesale changes in their output. We have maintained a simple correspondence between the text of an ALI program and the resulting layout so that changes in the layout can be easily related to changes in the program. This decision has simplified the system at the cost of making it less knowledgeable about MOS circuits.
- Facilitating the *division of labor*. Large layouts have to be produced by more than one designer. If the piece produced by each designer is specified in absolute positions, serious problems are likely to arise when the different pieces are put together, unless very tight interaction -- with its attendant penalties in productivity -- is maintained throughout the design. ALI allows the partitioning of tasks in such a way that the designer of a piece of the layout does not need to know anything about

the sizes of other pieces of the complete layout. For instance, on the top of fig. 1 three simple cells are shown with the intended connections between them shown by dotted lines; on the bottom of the figure, the pieces have been brought together to form a larger piece. The stretching that has taken place has occurred without the designer having to plan for it explicitly while considering each individual cell.

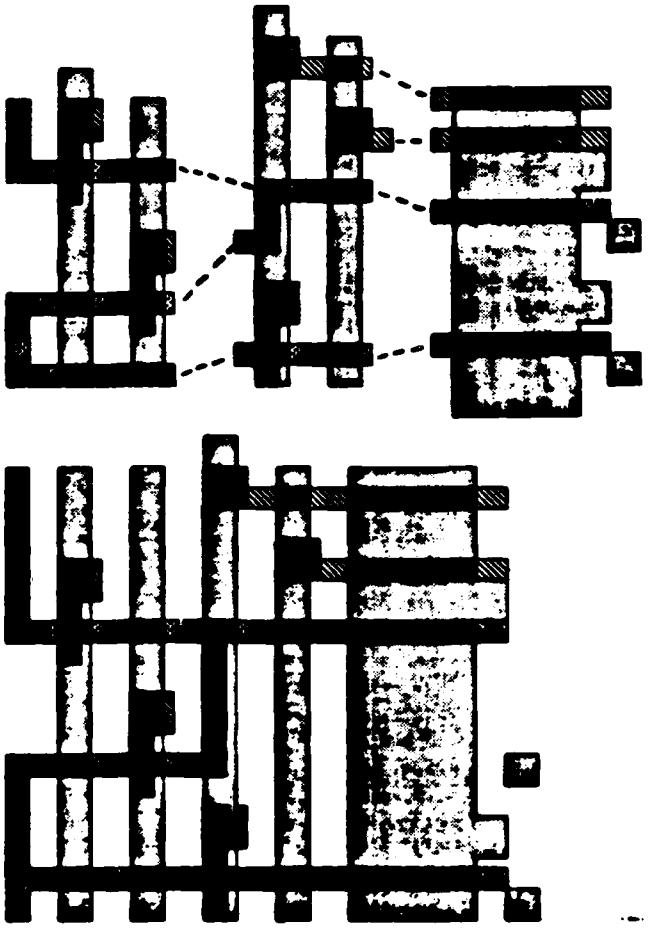


Fig. 1
Three separate cells and the result of connecting them along the dotted lines

- Facilitating *hierarchical design*. Even when a single designer is involved, the ability to view a layout as a hierarchy, with as much information about lower levels completely hidden from higher levels, is extremely useful. In ALI, the information about a given level of the hierarchy needed at the level immediately above is reduced by the absence of absolute sizes and positions, to topological relations among the layout elements of the lower level visible to the higher one.

- Reducing the ***life cycle*** cost of layouts. Modifying a layout to be fabricated on a new process, or to make it conform to a new set of design rules, is currently a costly operation. Yet successful designs seem to be more or less continuously updated as improved processes become available during their lifetime. Fig. 2 shows two instances of a simple layout produced with ALI. The instances are the result of running an ALI program twice changing ***exactly four constants in the program*** in between runs (those that specified the sizes of power and ground buses). This type of flexibility addresses the problem directly. An ALI program can be written naturally so that all layouts produced by it are completely free of design rule violations, no matter what the values of the constants used in the programs. Therefore the need for costly design rule checking of different instances of a layout (see fig. 2) can be avoided. The same ALI program can also generate layouts using different design rules by running it with a new module incorporating the new design rules.
- To avoid the ***need for special purpose computing equipment***. ALI can be used effectively from a standard ASCII terminal in combination with a small plotter shared by several designers. All the algorithms used in the inner cycle of ALI require ***linear time***, therefore permitting the use of just about any machine and guaranteeing fast turnaround on small layouts. Furthermore ALI replaces design rule checking by a hierarchical process that can be performed separately on the individual pieces of the layout. For example, after checking that each of the pieces shown on the top of fig. 1 is free of design rule violations, their combination shown on the bottom of the same figure will be guaranteed by ALI to be free of rule violations regardless of the stretch that takes place as a consequence of connecting them. ALI in fact requires far fewer computing resources than many design rule checking programs.

We feel that ALI succeeds in partially solving most of these problem. We do not claim however to have made the layout task trivial. To use a software metaphor, we feel that ALI elevates the work of the layout designer from absolute machine language programming, to programming in a relocatable assembler with subroutines. This not only makes the task more pleasant but makes new and more powerful tools possible such as loaders, linkers and compilers in the case of software. Similar tools for the VLSI world – which would indeed simplify the layout task enormously – remain, however, to be written. ALI should stand or fall with its ability to allow such tools to be built: whether we are right in believing that we have a framework in which these tools can be more easily implemented will not be known until our efforts in that direction succeed or fail.

The remainder of this section is devoted to a survey of the main features of ALI and a brief discussion of its current implementation.

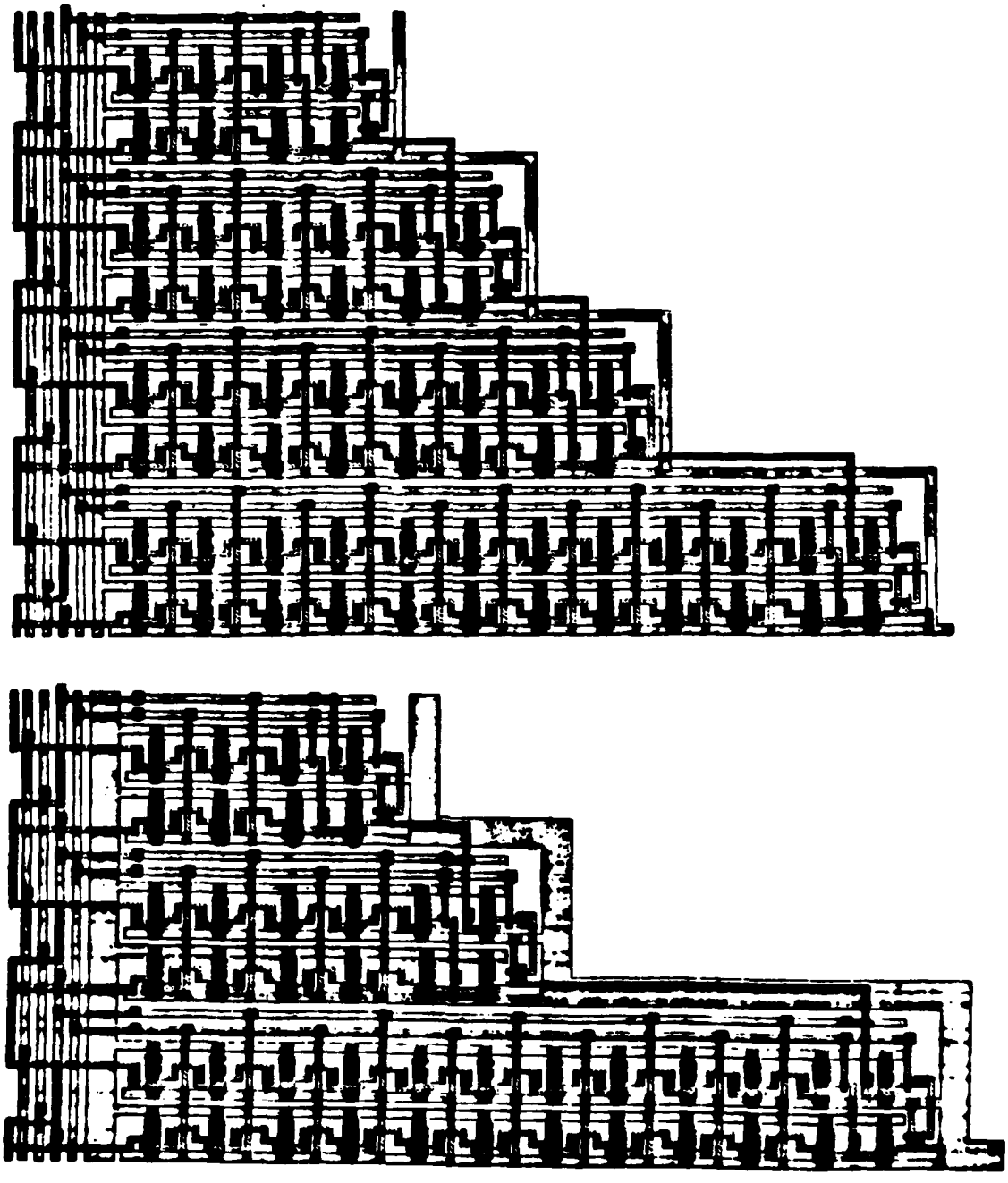


Fig. 2
Two ALI layouts generated by programs differing only in the values of four constants

2.1. An overview of ALI

The basic principles of ALI are quite simple. A layout is regarded as a collection of rectangular objects (with their sides oriented in the direction of the axes of a cartesian coordinate system) and a set of relations among these rectangles. The ALI user specifies a layout by declaring the rectangles (also called *boxes*) of which it is composed, and stating the relations that hold between them. ALI then generates a *minimum area* layout that satisfies all the relations between boxes specified in the program. For example, fig. 3 shows a trivial ALI program and the layout it produces.

```
chip simple;
  const
    hnumber = 10;
    length = 20;
    width = 6;
  boxtype
    htype = array [1..hnumber] of metal;
  var
    i : integer;
  box
    horizontal : htype;
    vertical : metal;
  begin
    for i := 1 to hnumber-1 do begin
      above ( horizontal[i], horizontal [i+1] );
      glueright ( horizontal[i], vertical );
      xmore ( horizontal [i], length )
    end;
    glueright ( horizontal[hnumber], vertical );
    xmore ( horizontal[hnumber], length );
    xmore ( vertical, width )
  end.
```

Fig.3
A simple ALI program and the layout it produces

This program looks very much like a Pascal program: it consists of a declarative part, followed by an executable part. To declare a box the user specifies its *name* (*horizontal* or *vertical* in the example), and its *type*, (*metal* – a predefined type – in the example). The standard box types correspond to the layers of the physical layout. As the example also shows, the ALI user can define structured objects (an array in the example). Further details on the type structure of ALI can be found in section 2.2.1.

The relations between the rectangles that make up a layout are specified in ALI through calls to a small set of *primitive operations* in the executable part. All such operations take as arguments boxes and possibly values of standard Pascal types (integers in our example). In our example *above*, *glueright* and *xmore* are primitive operations. The primitive *above* specifies that its first argument must appear above the second one in the final layout, the primitive *glueright* extends its first argument to the right to intersect its second argument, and *xmore* makes the size

of its first arguments along the *x* axis at least as large as the value of the second argument. Note that in this example ALI has determined the minimum separation between the horizontal elements as well as the minimum sizes of boxes not specified by ***xmore*** (such as the height of the horizontal metal lines) by accessing a table of design rules. More information about the type structure and the primitive operations of ALI is given in the next section.

When an ALI program is executed it generates two kinds of information. It produces a set of linear inequalities involving the coordinates of the corners of the boxes in the layout as variables. These inequalities, which embody the relations between the rectangles of the layout, are then solved to generate the positions and sizes of the layout elements. A brief description of the problems involved in this step can be found in section 2.3.2. The program also produces connectivity information about the rectangles in the layout. This information can then be used by a switch level simulator that predicts the behavior of the circuit as laid out whithout having to perform the ususal "node extraction" analysis.

In order for the layouts produced by an ALI program to be free of design rules, the program must be ***complete***, in that every pair of rectangles in it must be related in some way. Two rectangles may be related explicitly in the user program by virtue of being arguments to a primitive operation, or they may be related through the transitivity of the separations. The reason for this strong requirement is to prevent the area minimization process from shoving together rectangles that were intended to be separate (see section 2.3.3 fo a discussion of completeness).

ALI helps the designer to achieve this goal by generating certain relations between layout elements in an automatic fashion, and by checking on request whether this condition is satisfied. It is however the responsibility of the user to make an ALI program complete in this sense, as the computational cost of doing any sophisticated inference (beyond the transitivity of relations such as ***above***) is prohibitive [16].

2.2. Main features of ALI

This section describes how ALI appears to its user. Its three subsections deal, in turn, with the ***type structure***, the ***primitive operations*** of the language and the ***cell mechanism***. ALI has been built on top of Pascal and has inherited most of its features. In the interest of shortening this section we have assumed a certain familiarity with the general features of Pascal.

2.2.1. Type structure

As the example of fig. 3 shows, the objects manipulated by ALI are declared by stating their ***name*** and their ***type***. The types of ALI have the same structure as the Pascal types. Objects can be of a ***simple type*** (boxes) or of a ***structured type***.

There are a small number of ***standard types***, all of them simple. The standard types correspond to the layers of the process to be used to fabricate the layout (***metal***, ***poly***, ***diff***, ***impl***, ***cut*** and ***glass*** in the NMOS version currently implemented) plus the type ***virtual***, used to name bounding boxes and having no physical reality in the fabricated circuit. For example, in the program of fig. 2, the declaration

vertical : metal

specifies that the rectangle named ***vertical*** on the final layout should be on the metal layer. ALI will use this information to generate constraints on its minimum size and its separation from other layout elements.

Structured types are of two flavors: ***array*** (a collection of objects of the same type) and ***bus*** (a collection of objects of heterogeneous types, much like *records* in Pascal), which correspond directly to the array and record structured types of Pascal. ALI, like Pascal, permits the creation of new ***user defined*** types that can be either simple or structured. For example, in fig. 3, the fragment

htype = **array** [*1..hnumber*] *of metal*

inside the **boxtype** section of the program, creates a new type, *htype*, each object of that type made up of a number of metal rectangles, and the fragment

horizontal : htype

inside the **box** section, creates an object of that type named *horizontal*.

In a similar fashion the type declaration

```
shiftbus = bus
    ph1, ph2 : metal;
    vdd : metal;
    data : diff;
    gnd : metal
end
```

creates a user defined type, allowing the user to create objects which consist of four metal boxes and a diffusion box. The types of the components of structured types are arbitrary: the user can define arrays of buses, or buses containing arrays.

The accessing of the elements of arrays and buses is done as in Pascal. Thus if x is of type *htype* then $x[i]$ refers to the i-th element of x, and if y is of type *shiftbus* then *y.data* refers to the diffusion box of y.

Although the structured objects are generally used by ALI simply as a naming mechanism, they are also used in conjunction with the cell mechanism (discussed in section 2.2.3) to automatically generate separations between boxes. We will be more precise on this point when we describe the cell mechanism of ALI.

Like Pascal, ALI is a ***strongly typed*** language. The primitive operations know about certain type restrictions and generate type mismatch errors if operations are attempted with rectangles of inappropriate types.

2.2.2. Primitive operations

The relations between the rectangles that make up a layout are specified in ALI through calls to a small set of primitive operations. All such operations take boxes (i.e., objects of simple types) as arguments. In the program of fig. 3, ***above***, ***glueright*** and ***xmore*** are primitive operations.

It is not important to know the actual primitive operations of the current version of ALI to understand its operation. As a gross measure of its complexity we can say that the system currently implemented -- based on NMOS as described in [12] -- has about twenty primitive operations which can be arranged in the following groups:

1 *Separation primitives*: such as ***above*** in fig. 3, which specify that their arguments must be separated in a certain direction in the final layout. The minimum amount of space between boxes separated in this manner depends on their types and is supplied by ALI from a table of design rules.

2 *Connection primitives*: such as ***glueright*** in fig. 3, to specify that their arguments -- which must be boxes in the same layer -- are to be joined in a particular manner.

3 An *inclusion* primitive, ***inside***, that specifies that one box is to be placed inside another. The minimum distances between their edges are again suplied by ALI from a table of design rules.

4 *Minimum size primitives*: such as ***xmore*** in fig. 3, which specify the minimum size of a box along a certain direction. Default minimum sizes are provided by ALI from a design rule table.

5 *Transistor primitives*, which create depletion mode and pass transistors.

6 *Contact Primitives*, which create contacts between layers and connect boxes to them.

Note that no absolute positions or dimensions for any rectangle can be specified with these primitives. All the rectangles of a layout can be stretched and compressed (up to a minimum size) and all can float in any direction. If one single characteristic is to be used to separate ALI from other layout systems, this must be it. Most of the power of ALI and most of the problems one faces in its implementation are consequences of this fact.

It is important to remember that in order for a layout produced by ALI to be free of design rule violations, any two rectangles in it must be related in some way. ALI will make no inferences as to the relations between boxes beyond those implied by the transitivity of some primitive operations (i.e., if ***above*** *(a, b)* and ***above*** *(b, c)* are stated, ***above*** *(a, c)* need not be stated). Although the system generates a good number of

relations automatically for the user, particularly in connection with the cell mechanism (see the next subsection), there is still a fair amount of drudgery left for the user in making sure that this requirement is met. A brief discussion on the computational complexity of the automatic generation of relations between boxes can be found in section 2.3.3.

2.2.3. Cells

Perhaps the most powerful feature of ALI is its procedure-like mechanism for the definition and creation of *cells*. A cell is a collection of related rectangles enclosed in a rectangular area. Rectangles that are inside a cell are of two types: *local* which are invisible to the outside, or *parameters* which can interact in a simple and well defined manner with rectangles outside the cell.

A cell is *defined* by specifying its local objects, its formal parameters and the relations among all of them. Once a cell has been defined, it can be *instantiated* as many times as desired by specifying the actual parameters for the instance, much the same way as one invokes a procedure or function in a procedural language. The result of instantiating a cell is to create a brand new copy of the prototype described in the cell definition with the formal parameters connected to the actual parameters.

A cell definition is made up of a *header*, in which the formal parameters are described, a set of *local box declarations* and a *body* in which the relationship between the parameters and the local boxes, as well as those among local boxes, are specified.

The header describes the names and types of the parameters and the side of the bounding rectangle through which they come into contact with the inside of the cell. The header of a cell (using the type *shiftbus* defined in section 2.2.1) and an instance of it are shown in fig 4.

cell *shift* **(left** *l* **:** *shiftbus*; **right** *r* **:** *shiftbus* **)**

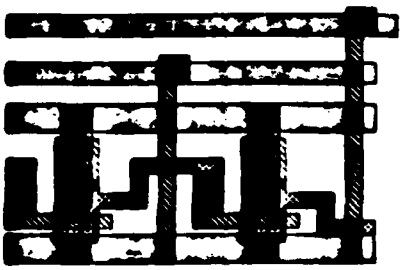


Fig. 4
A sample cell definition header and an instance of the cell defined

Cells may have any number of parameters on each of their four sides. The order in which they are listed in the cell header describes their relative positions. Horizontal parameters (i.e., those touching the cell on the left or right) are assumed to be listed in top to bottom order and vertical parameters in left to right order.

The body of a cell is very much like an ALI program. For example, fig. 5 shows a complete cell definition that consists of a variable number of *shift* cell instances connected sequentially together with two of its instances. Note that cells are instantiated by the **create** statement, and that the parameter list of the cell contains both box parameters and other parameters (an integer in this case) in separate lists. Note also that recursion has been used to define this cell; this highlights the fact that ALI has the full power of Pascal at its disposal.

When an instance of a cell is created it can be given a name, provided that the name given has been declared as a rectangle of the standard simple type *virtual*. The relationship of the rectangle bounding a newly created cell to any other rectangle of the layout can be specified in the standard manner by calls to the primitive operations. This is a vital feature since in many cases (i.e., *above*, *below*...) stating a relation between two cell instances c_1 and c_2 immediately implies a relation between every pair of rectangles r_1 and r_2 such that r_1 is part of c_1 and r_2 part of c_2.

There are two important ways in which the cell mechanism helps in the automatic generation of constraints between boxes. When an object of a structured type is passed as a parameter to a cell, its component boxes are separated from top to bottom (if it is a **left** or **right** argument) or from left to right (if it is a **top** or **bottom** argument). The order of the separation is determined by applying recursively the following rules: array elements are separated ordered by their indices and bus elements in the order in which they were specified in the bus declaration. Thus, in the example of fig. 5, the components of parameter *inbus* would be separated from top to bottom. The second mechanism involves the automatic separation of cells that share a parameter; thus in the example of fig. 5, the individual instances of *shift* are separated automatically, since adjacent instances share a parameter.

The cell mechanism gives the ALI user the ability to describe layouts in a truly hierarchical manner. A proper ALI design, very much like a well structured program, will consist of a hierarchy of cell instances with only a small amount of information at a given level (the parameters of the cell instances at that level) being visible from the immediately higher level. For example, the layout given in fig. 2 consists of four instances of the same cell stacked vertically. That cell in turn is defined in terms of three other cells, one of them being the cell shown in fig. 1, which is in turn defined in terms of three other cells.

Much of the power and generality of the cell mechanism of ALI comes from the absence of absolute positions and sizes in a layout specification. In particular, two instances of the same cell may have radically different sizes depending on the actual parameters used to create them, as exemplified by figs. 1, 2 and 5. We believe that no cell mechanism can be

```
cell shiftregister (   left   inbus : shiftbus;
                       right outbus : shiftbus )
                   ( length : integer );
  box
    temp : shiftbus;
  begin
    if length = 1 then
      create shift ( inbus, outbus )
    else begin
      create shift ( inbus, temp );
      create shiftregister ( temp, outbus ) ( length -1 )
    end
  end;
```

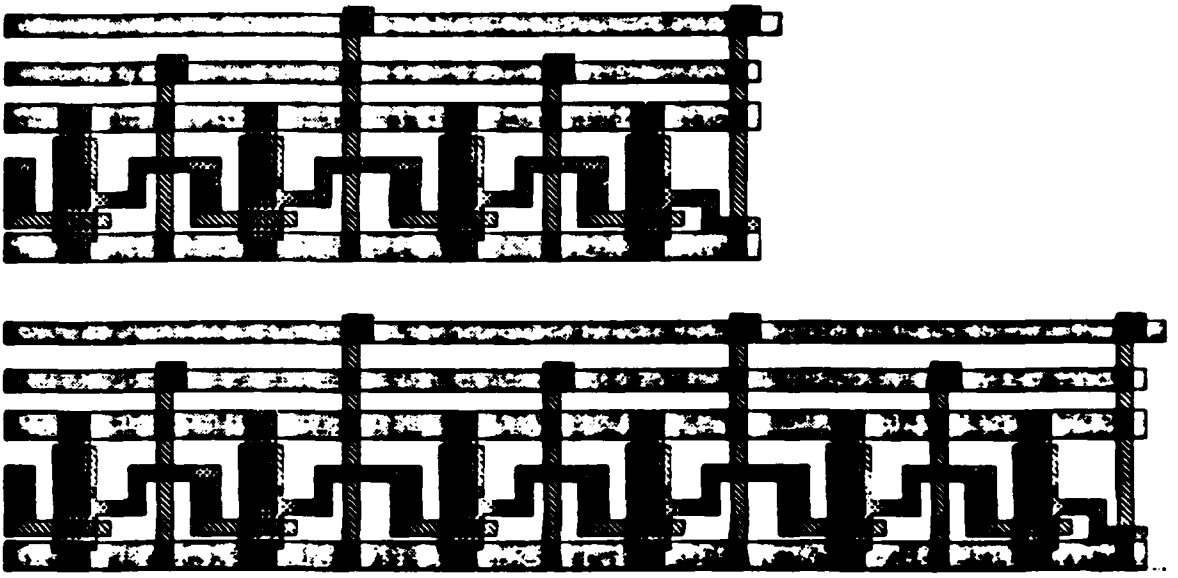


Fig. 5
A cell definition and two instances of it generated by a simple ALI program

said to be truly general unless the sizes of its arguments and local rectangles, as well as their relative distances are determined at the time the cell is instantiated.

There are some penalties involved in the use of the cell mechanism. In particular, ALI generates separations between cells in a manner which is oblivious to what is inside them. That is, the minimum separation between cells as far as ALI is concerned, is the maximum of all the minimum separations for two layers in the design rules, thus creating a certain wastage. We believe that this penalty will be generally a small percentage of the total area and is well worth the advantages gained by

the ability to separate cell instances as units.

Another source of wastage is the fact that cells are restricted to be bounded by a rectangle, so the packing of cells that have irregular shapes results in a certain amount of unused space. The rectangular shape of the cells is a fundamental characteristic of ALI: The introduction of irregularly shaped cells is simply not possible without completely redesigning the language and. However, the waste introduced because of this restriction can be avoided in most particular cases through some code modifications.

2.3. Implementation issues

The previous section described the user view of ALI. In this section we discuss briefly some of the problems to be solved when trying to go from an ALI program to a layout that satisfies the relations stated in it. We first give an overall description of the system as currently implemented, then discuss the method used to assign locations and sizes to the layout elements and then the concept of *completeness* and how it is checked.

2.3.1. Overall implementation

The current version of our system has been implemented as follows. The ALI program is first translated into standard Pascal. The resulting Pascal program is then compiled and linked with a precompiled set of procedures that implement the primitive operations and the resulting object module is then run. The output of this program (generated entirely by the primitive operations) is a set of linear inequalities and connectivity relations among the layout elements. The inequalities are then solved to generate a layout or examined by a program that checks their logical completeness, and the connectivity information can be used to simulate the circuit laid out.

The design rules are incorporated as a table which is used by the primitive operations to produce the linear inequalities. Thus changing the design rules for our system requires only to change this table.

2.3.2. Placement

As explained above, one of the results of running an ALI program is a set of linear inequalities that embody the relations between the layout elements. These inequalities are of the following simple form:

$$x_i - x_j \geq d \quad (d \geq 0)$$

where the variables are the coordinates of the corners of the boxes that form the layout and the constants are either user supplied (as in the second argument of the *xmore* primitive, for instance) or extracted from the table of design rules by the system itself.

This set of inequalities should be solved so as to generate placements for the boxes that compose the layout in such a way as to minimize its total area. In order to perform this task efficiently, we require that no inequality in the set involve both x and y coordinates. This restriction allows us to minimize the total area by minimizing the maximum x and y coordinates of any point independently, at the cost of reducing the range of the relations between boxes that we can express. We cannot, for instance, handle rectangles whose sides are not parallel to the cartesian axes or express aspect ratios directly.

We have now a sufficiently simple problem so it can be solved in *time proportional to the number of inequalities in our set*. (All layouts that can be expressed in ALI can be generated by a program that produces a constant number of inequalities per rectangle). This is done by a version of the topological sort algorithm [11] applied to the x and y coordinates independently. This algorithm assigns to each point the lowest possible coordinate while minimizing the largest coordinate of all points.

The form of the inequalities that we allow is rather restrictive; it is sufficient however, to describe the design rules given in [12] for NMOS, and the efficiency gained in return for this simplicity seem to us like a good tradeoff. A more subtle consequence of the simplicity of the inequalities and the method we use to solve them is that undesirable stretching can occur, since we have no way to specify a maximum size for any object. This is not a common occurrence and the user can in all cases guard against such stretching by the careful selection of the primitive operations used. It is nonetheless an additional burden placed on the designer.

The choice of an efficient placement algorithm over expressibility power and a reduced degree of user convenience has been quite conscious in this particular case. We feel that every reasonable measure should be taken to keep the complexity of the placement problem linear, given that the size of layouts is currently large (10^7 rectangles) and is growing fast. Widening the class of linear inequalities acceptable is almost certain to make linear time solutions impossible [2].

2.3.3. Completeness

ALI programs do not involve absolute sizes or positions of boxes, and are, to a great extent,independent of the design rules. These characteristics make it clearly desirable to insure that the layout described by a program will be free of design rule violations in a way other than checking the finished layout. The following paragraphs describe a way of insuring freedom from design rule violations in a manner that is independent of the actual design rules used to generate the final placement. The description may be somewhat cryptic; the interested reader is referred to [16] for further details.

A layout generated by an ALI program is *complete* if for any two boxes a and b whose types make it possible for them to interact in the final layout, either

(i) a and b are explicitly stated to be in contact by some primitive operation, or

(ii) a and b are, explicitly or through the transitivity of primitive relations, stated to be separated in either the x or the y direction by a minimum amount which depends on their types.

From this definition, it should be clear that testing completeness of a cell instance involves computing the transitive closure of a graph. Therefore the complexity of the operation will be $O(n^3)$, where n is the number of boxes in the cell. It is thus not feasible to test a large layout for completeness in a direct way.

Fortunately, completeness can be checked hierarchically. Let us look only at the objects at the highest level of the hierarchy of boxes that defines a layout, i.e., those boxes (including cell boundaries) defined globally in the ALI program that generated the layout. If these objects are related in a complete manner and the cell instances used at this level are also complete, then the whole layout is complete.

Thus one can check the completeness of a layout by successively checking cell instances for completeness, thereby reducing the complexity of the process to $O(m^3)$ where m is the largest number of boxes local to a cell instance in the layout. This process can be reduced further, since not every cell instance needs to be checked. For example, if a cell is defined by a straight line program, checking one instance for completeness suffices, as one instance of the cell will be complete if and only if all of its instances are [16]. The case of cells with branches and iteration is not quite as simple. Yet we are confident -- and our experience tends to confirm this belief -- that checking the completeness of a few carefully selected instances of any cell definition will be enough to guarantee that the cell definition is complete.

The end result is that completeness has the flavor of a static, almost syntactic, property for all non malicious examples, and is much easier to check in a well structured layout than design rule freedom by the standard means on the final layout.

Finally, a word about the possibility of taking an incomplete layout specification and automatically completing it. The general problem of generating an optimal completion is NP-Complete, but the simpler version of generating any completion for graphs embedded in a grid (as our layouts are) seems to be solvable in $O(n^2)$ steps. The question of how much area will be wasted by such a completion algorithm will have to wait for some experimentation, but there is no question of its usefulness.

2.4. Experience with ALI

The current implementation of ALI has shown the soundness of most of our original ideas. The system is efficient and the language easy to learn, and the layout it produces are relatively dense (for example, an ALI program written without concern for area optimization produced a layout which was about thirty per cent larger than a similar layout packed by

hand on a graphics editor. Unfortunately, this evidence has been gathered mostly from people who had a hand in designing or implementing ALI. Perhaps a more reliable evaluation of ALI, ought to wait until a substantial number of users not involved in its design can give an informed opinion. We hope to obtain this evidence before long, since ALI is currently being used in a VLSI design course.

The fact that very little effort was invested in error recovery for the sake of expediency in getting a prototype running, and that no mechanism for integrating separately produced layout pieces was provided make the current system useful mostly for teaching purposes and experimentation. It must be emphasized that this is a result of implementation choices, and not of any intrinsic limitation on the approach we have taken.

The problems of the current system which we plan to address with the next version are the following:

(1) Memory requirements. The solution of the system of linear inequalities requires large amounts of memory. We will use a different algorithm which is slightly less efficient in terms of time but requires an order of magnitude fewer memory locations for a typical large layout.

(2) Pascal problems. The current ALI has exactly the same type structure as Pascal. The lack of generic types and dynamic arrays has made the task of writing general purpose tools (PLA generators, routers...) inside ALI more difficult than it ought to be. The next ALI will have the notions of generic types and dynamic arrays.

(3) Connecting primitives. Certain objects, such as contacts, are used frequently enough to warrant making them part of the language.

(4) Separate "compilation" facilities. Clearly, large layouts will have to be generated in pieces, which is something that our current system cannot do.

Acknowledgements

We would like to thank Jose Mata, Vijaya Ramachandran and Jerry Spinrad for their help in the implementation of ALI, and Jean Vuillemin, Scot Drysdale and the referees of the paper for their comments. We also want to thank Bruce Arden for his advice and support.

The work of Richard Lipton has been partially supported by grants MCS8023-806 from NSF and N00014-82-K-0549 from DARPA and ONR. Stephen C. North is being supported by Bell Laboratories. Robert Sedgewick's work was partially supported by NSF grant MCS80-17579. The work of Jacobo Valdes has been supported by DARPA and ONR grant N00014-82-K-0549.

3. References

[1] Ackland, B., Weste, N., "A pragmatic approach to topological symbolic IC design. design," *VLSI'81*, pp 117-129, John P. Gray ed., Academic Press.

[2] Apsvall, B. and Shiloach Y., "A Polynomial Time Algorithm for Solving Systems of Linear Inequalities with Two variables per Inequality", pp 205-217, *Proc. of the twentieth IEEE Symp. on Foundations of Computer Science*, 1979.

[3] Baker, C. M., "Artwork Analysis Tools for VLSI Circuits," M. S. Thesis, MIT, EECS Department, June, 1980.

[4] Batali, J., Mayle, N., Shrobe, H., Sussman, G., Weise, D., "The DPL/Daedalus Design Environment," *VLSI'81*, pp 183-192, John P. Gray ed., Academic Press.

[5] Bryant, R. E., "MOSSIM: A switch-Level Simulator for MOS LSI," pp 786-790, *18th Design Automation Conference*, 1981.

[6] Davis, T., Clark, J., "SILT: A VLSI Design Language (Preliminary Draft)", unpublished manuscript, Digital Systems Laboratory, Stanford University.

[7] Eichemberger, P., "Lava: an IC layout language", unpublished manuscript, Electronics Research Laboratory, Stanford University.

[8] Fairbairn, D., Rowson, "Icarus: an Interactive Integrated Circuit Layout Program", pp 188-192, *15th Design Automation Conference Proceedings*, 1978.

[9] Fan, S. P., Hsueh, M. Y., Newton, A. R., Peterson, D. O., "MOTISC: A new circuit simulator for MOSLSI circutis," *IEEE Proc. Int. Symp. Circuits and System*, pp 700-703, 1977.

[10] Franco, D., Reed, L., "The Cell Design System", pp 240-247, *18th Design Automation Conference Proceedings*, 1981.

[11] Holt, D., Shapiro, S., "BOLT -- A Block Oriented Design Specification Language", pp 276-279, *18th Design Automation Conference Proceedings*, 1981.

[12] Johannsen, D., "Bristle-Blocks", pp 310-313, *16th Design Automation Conference Proceedings*, 1979.

[13] Johnson, S. C., "The LSI Design Language i", unpublished manuscript.

[14] Knuth, D. E., *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, Addison-Wesley, 1971.

[15] Mead, C., Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[16] Mosleller, R.C., "REST: A leaf cell design system," *VLSI'81*, pp 163-172, John P. Gray ed., Academic Press.

[17] Sastry, S., Klein, S., "PLATES: A Metric Free VLSI Layout Language", pp 165-169, *Proceedings of the 1982 Conference on Advanced Research in VLSI*, 1982.

[18] Tarjan, R. E., "Efficiency of a Good but Not Linear Set Union Algorithm", *JACM*, vol. 22, no.2, pp 215-225, 1975.

[19] Trimberger, S., "Combining Graphics and a Layout Language in a Simple Interactive System," *18th Design Automation Conference Proceedings*, 1981.

[20] Vijayan, G., "Completeness of VLSI Layouts", VLSI Memo #1 Princeton University, Department of Electrical Engineering and Computer Science, September 1982.

[21] Williams, J., "STICKS, A Graphical Compiler for High Level LSI design", pp 289-295, *Proceedings of the 1978 NCC*, 1978.

Rectilinear Graphs and their Embeddings

Gopalakrishnan Vijayan
Avi Wigderson

Department of Electrical Engineering and Computer Science
Princeton University,
Princeton, New Jersey 08544

1. Introduction

The problem we address in this paper is an embedding problem for a class of graphs which we call rectilinear graphs. These graphs are important in many VLSI layout problems. In fact, this problem arose in the implementation of ALI [6,7], a procedural language for VLSI design currently under development at Princeton. An embedding algorithm can be used to automate the production of VLSI layouts in many procedural design systems.

Consider the following model for VLSI layout design. A VLSI layout is described hierarchically using cells and wires that connect the cells together. Each cell C is enclosed within a rectangle $R(C)$, and has four tuples of pins, one each for the left, top, right, and bottom of rectangle $R(C)$. Each wire w is denoted by a pair of pins (p_i, p_j), such that p_i and p_j are pins of different cells and are of opposite types. For example, if p_i is a right pin then p_j should be a left pin. Given such a description of a VLSI layout, our aim is to produce an embedding of the description on the plane, such that (*i*) no two bounding rectangles touch each other, (*ii*) the pins appear in the correct order on the bounding rectangles, (*iii*) the wires are straight and rectilinear, and (*iv*) no two wires cross each other. Later on, we can fill each bounding rectangle $R(C)$ with the embedding of the cell C in the same manner.

The restriction that wires cannot be bent may seem unrealistic, but this is certainly the case in many design systems including ALI. If a wire has to be bent, the user specifies that by breaking up the wire into several straight wires and placing cells at each of the turn points of the wire. In ALI, for example, the user can incorporate routing algorithms in a ALI program to determine how the wires are to be bent. The restriction that wires cannot cross implies that we are dealing with the wires on a single layer. For a layout with multiple layers, it is clearly necessary that the wires on each layer do not cross.

To solve the above embedding problem, it is enough to consider only a simple restriction, where each bounding rectangle has at most one pin on each side. We can then treat the bounding rectangles as vertices and the wires as edges, which leave in one of the four cardinal directions. We give the name *rectilinear*

graphs to such graphs. The embedding problem of rectilinear graphs is our main concern in this paper.

For VLSI applications, we need efficient algorithms to recognize and then actually embed rectilinear graphs. In this paper, we present an $O(n)$ recognition algorithm and an $O(n^2)$ embedding algorithm, where n is the number of vertices in the graph. Thus, a hierarchically described VLSI layout with cell instances $C_1, C_2, \cdots, C_m$ can embedded in time $O(\sum_{i=1}^{m} n_i^2)$, where n_i is the number of pins in cell instance C_i.

An embedding of a rectilinear graph is just a relative placement of the vertices (cells) on a rectangular grid, such that no two edges cross. Some of the relative placement information is already present in the description of a rectilinear graph. For example, if (a,b) is the rightgoing edge of vertex a, then a should be to the right of b, and a,b should be on the same horizontal grid line. Hence, an embedding can be viewed as a "completion" of the rectilinear graph description. We showed in a different paper [9] that the completion problem for a slightly more relaxed VLSI layout model is NP-complete. In light of this result, the results in this paper have become more important.

In section 2, we present formal definitions of rectilinear graphs and their embeddings. In section 3, we mention some properties of rectilinear graphs. We discuss some topological properties of the embeddings in section 4. A necessary and sufficient condition for biconnected rectilinear graphs to be embeddable is presented in section 5. A similar condition for arbitrary rectilinear graphs is the main result in section 6. We also describe a $O(n)$ recognition algorithm in this section. In section 7, we use the ideas of the previous sections to obtain an $O(n^2)$ embedding algorithm. An important subclass of rectilinear graphs is discussed in section 8. In section 8, we discuss extensions and open problems. For definitions of graph theoretic terminology used in this paper, please refer to [1,2].

2. Definition of the Problem

First we give a formal definition of a rectilinear graph.

Definition 2.1: A ***rectilinear graph*** G is a triple (V, E, λ), where V is the vertex set, E is the edge set, and

$$\lambda : V \times V \rightarrow \Sigma \cup \{\varepsilon\}, \text{ where } \Sigma = \{L, R, D, U\}$$

is a ***vertex ordering relation*** with the following properties:
for every $a,b,c \in V$ and $X \in \Sigma$

(i) $\lambda((a,b)) = \varepsilon \Leftrightarrow \{a,b\} \notin E$

(ordering is specified only between adjacent vertices)

(ii) $\lambda((a,b)) = L \Leftrightarrow \lambda((b,a)) = R$, $\lambda((a,b)) = D \Leftrightarrow \lambda((b,a)) = U$,

(iii) $\lambda((a,b)) = X \rightarrow \lambda((c,b)) \neq X, \forall c \neq a$ (no overlapping edges).

Each vertex in a rectilinear graph has degree at most four, and each edge (a,b), as it goes from one vertex a to the another b, has a nonempty label on it, which in the embedding will indicate the direction (left,right, down, or up) in which the edge leaves vertex a. There can be at most one edge with a particular label emanating from each vertex. The undirected graph $G(V,E)$ will be referred to as the ***underlying graph***. Figure 2.1 (like all other figures) gives an illustration of a rectilinear graph.

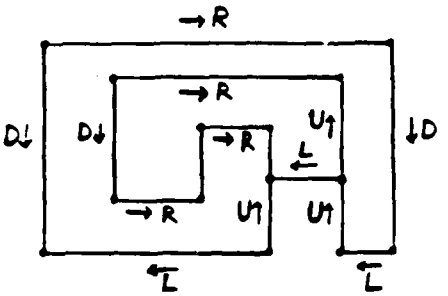


Figure 2.1
A rectilinear graph

Now we define what sort of an embedding we are looking for.

Definition 2.2: An ***embedding*** of a rectilinear graph $G(V,E,\lambda)$ on a ***rectangular grid*** is given by two mappings $x, y: V \rightarrow Z$ (the integers) which are the x and y coordinates respectively of the vertices. These mappings obey:

1. **the ordering relation**, λ, i.e. for all edges $\{a,b\} \in E$

$$\lambda((a,b)) = L \rightarrow y(a) = y(b),\ x(a) > x(b),$$

$$\lambda((a,b)) = R \rightarrow y(a) = y(b),\ x(a) < x(b),$$

$$\lambda((a,b)) = D \rightarrow x(a) = x(b),\ y(a) > y(b),$$

$$\lambda((a,b)) = U \rightarrow x(a) = x(b),\ y(a) < y(b).$$

2. **Planarity**, no two edges cross, i.e. for each pair of non-adjacent edges $\{a,b\}, \{c,d\}$ such that $\lambda((a,b)) = R$ and $\lambda((c,d)) = U$, the condition

$$x(a) \leq x(c) \leq x(b) \text{ and } y(c) \leq y(a) \leq y(d)$$

does not hold.

An embedding of a rectilinear graph on a rectangular grid is one in which the vertices are placed at grid points, the edges run along grid lines in the directions given by their labels, and no two edges cross each other except if they share a vertex. We say that a rectilinear graph is embeddable if it has an embedding. We will show in the next section that not all rectilinear graphs are embeddable.

Now our main problem can be stated simply: Given a rectilinear graph $G(V,E,\lambda)$, is it embeddable, and if yes, find an embedding.

3. Some Comments on Rectilinear Graphs

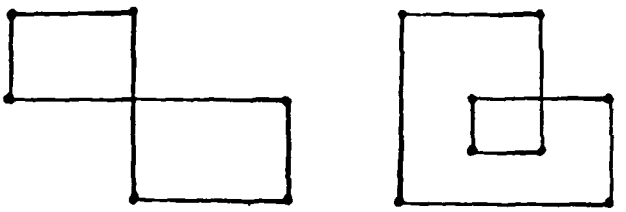


Figure 3.1
Two nonembeddable rectilinear cycles

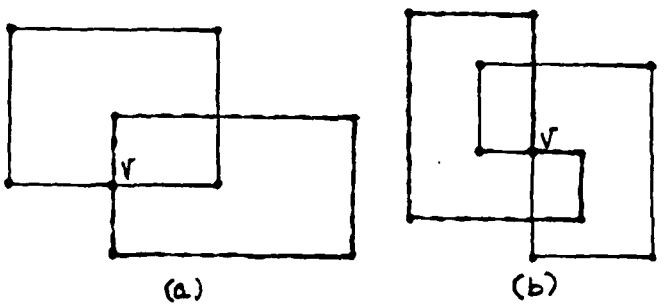


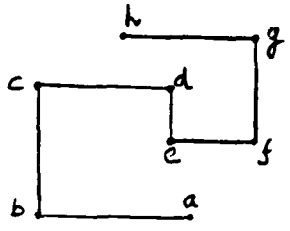
Figure 3.2
Two nonembeddable rectilinear graphs
whose biconnected components are

In this section we list some properties of rectilinear graphs and their embeddings. These will give an indication of why the problem is not trivial and why it is different from other embedding problems, and in particular, planar graph embedding [3,5].

1. Embeddability is a hereditary property. Subgraphs are defined in the usual fashion, but here the labels of edges are inherited. This is obvious, but worth mentioning, because this will be used in the proofs.
2. If each connected component of a rectilinear graph is embeddable then the graph itself is embeddable. So, without loss of generality we will restrict ourselves to connected rectilinear graphs.
3. Rectilinear graphs with nonplanar underlying graphs are clearly not embeddable. So it is not interesting to consider those graphs. However, not every rectilinear graph with a planar underlying graph is embeddable. In figure 3.1, we have two simple cycles which are not embeddable.

4. In contrast with planarity, embeddability is *not* a property determined by the biconnected components. Figure 3.2 provides an illustration of this fact.
5. This problem is a restriction of an NP-complete problem [9,11]. For each wire w, we are given its orientation (horizontal or vertical), and a set V_w of vertices. The wire w has to pass through each vertex in the set V_w (the vertices could be touched in any order). Then, the embedding problem becomes NP-complete.
6. If we relax the rectilinearity of the edges and impose only the cyclic ordering of the edges at each vertex, then there is an $O(|V|)$ algorithm [10]. The cyclic orderings automatically determines the faces of the embedding (if one exists). Thus a embeddable rectilinear graph has a unique embedding in this sense.

4. Topological Structure of Embeddings



$\lambda((abcdefgh)) = LURDRUL$

Figure 4.1

The extension of λ to paths.

There is a natural way to extend the function λ to paths and cycles in the graph as follows. Given a path $P = (v_0, v_1, \cdots, v_t)$ we define $\lambda(P) = \lambda((v_0,v_1))\lambda((v_1,v_2)) \cdots \lambda((v_{t-1},v_t))$. We define a similar extension for cycles where now $v_t = v_0$. λ becomes a mapping that associates with each path or cycle in the graph a string in Σ^* which is the concatenation of labels along the path or cycle. Note that strings containing RL, DU, LR, UD as substrings do not represent paths. Also the direction in which we traverse a path and the starting point in a cycle are important. An example of this mapping can be found in figure 4.3.

Next we define two topological actions on rectilinear graphs. These actions will simplify a rectilinear graph while preserving its topological structure. Let G be a rectilinear graph.

Action 1 - Edge Contraction: Let $(abcd)$ be a path in G such that both b and c have degree 2, and $\lambda((abcd)) = XYX$ where $X, Y \in \Sigma$. Contract the edge (b,c) to the vertex b. The resulting path (abd) will have $\lambda((abd)) = XX$. We abbreviate

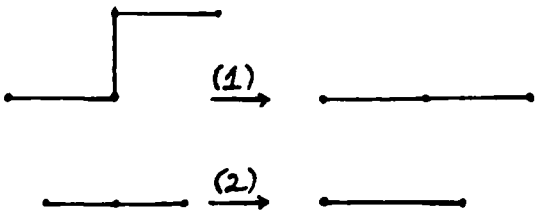


Figure 4.2
Edge contraction and vertex deletion

this action by $XYX \rightarrow XX$ (figure 4.2(1)).

Action 2 - Vertex Deletion: Let (abc) be a path in G such that vertex b has degree 2, and $\lambda((abc)) = XX$ where $X \in \Sigma$. Delete the vertex b and introduce the edge (a,c). The resulting edge (a,c) will have $\lambda((a,c)) = X$. We abbreviate this action by $XX \rightarrow X$ (figure 4.2(2)).

In a natural way we can define inverses for the above two actions which we will refer to as **edge expansion** and **vertex addition** respectively.

Lemma 4.1: Let G be a rectilinear graph and G' be the graph resulting from G by the application of a sequence of the above four actions. Then G' is also rectilinear and moreover G' is embeddable if and only if G is embeddable.
Proof: The proof is easy and is left to the reader. •

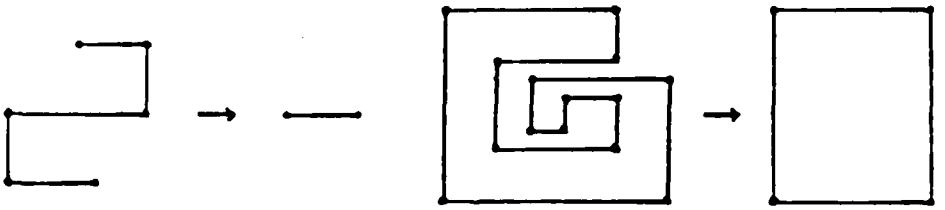


Figure 4.3
Simplification of a path and a cycle

Definition 4.1: Given a string $\gamma \in \Sigma^*$ representing a path or a cycle, the ***simplified form*** $\bar{\gamma}$ ***of*** γ is obtained by repeatedly applying the reduction rules $XYX \rightarrow XX$ and $XX \rightarrow X$, where $X, Y \in \Sigma$, until they cannot be applied any more. If γ represents a cycle then it is treated as a cyclic string.

In figure 4.3 we give a path and a cycle along with their simplified forms.

Lemma 4.2: Every string $\gamma \in \Sigma^*$ has a unique simplified form.
Proof: The replacement system defined by the two reduction rules have the Church-Rosser property [8]. •

Definition 4.2: A ***square*** is one of the cyclic strings *LURD* or *LDRU*.

Sometimes we may distinguish between two squares by their starting labels.

Definition 4.3: A ***spiral*** is a path which cannot be simplified. Equivalently a spiral is a substring of $(LURD)^*$ or $(LDRU)^*$.

Lemma 4.3: Every path is embeddable.
Proof: Every spiral is embeddable. Since any path simplifies to a spiral (by definition), by lemma 4.1 it is also embeddable. •

So it is the cycles which make the problem nontrivial. The following lemma is a crucial fact about cycles.

Lemma 4.4: A cycle is embeddable if and only if it simplifies to a square.
Proof: ***if:*** A square is embeddable and hence by lemma 4.1 any cycle which simplifies to a square is also embeddable.

only if: Let f be an embeddable cycle and $\lambda(f) = \gamma$. By lemma 4.1, the cycle defined by $\bar{\gamma}$ is also embeddable. Let $|\gamma| = n$. Look at the embedding of $\bar{\gamma}$. Since it has no crossings the embedding is a simple polygon. Therefore the interior angle of this polygon sum to $(n-2).180°$. Since $\bar{\gamma}$ is a spiral all its interior angles are 90°. The only solution to $n.90 = (n-2).180$ is $n = 4$. Therefore $\bar{\gamma}$ is a square. •

The proof of the previous lemma suggests another useful characterization of embeddable cycles. Going along a cycle $f = v_1 v_2 \cdots v_n v_1$ in the counterclockwise direction, let us denote by $\varphi_f(v_i)$ the angle at vertex v_i, which is the angle between (v_{i-1}, v_i) and (v_i, v_{i+1}).

Lemma 4.5: A cycle $f = v_1 v_2 \cdots v_n v_1, n \geq 4$ is embeddable if and only if $\varphi(f) = \sum_{i=1}^{n} \varphi_f(v_i) = (n \pm 2).180°$.

Proof: Suppose f is embeddable, then its embedding is a simple polygon. Depending on whether we sum the interior angles or exterior angles we should get $(n \pm 2).180°$.

To prove the sufficient part we show by induction of n that f simplifies to a square. The possible values for $\varphi_f(v_i)$ are 90,180,270°. The basis for induction is $n = 4$. In this case the given sum of the angles is either 360° or 1080°, which implies that each angle is either 90° or 360° respectively. So f must be a square by itself.

Assume that the claim is true for all values less than n and let $n > 4$. If for

some i, $\varphi_f(v_i) = 180°$, then $\lambda(v_{i-1}v_iv_{i+1}) = XX$. We can apply vertex deletion at v_i to obtain $f' = v_1v_2 \cdots v_{i-1}v_{i+1} \cdots v_nv_1$. Then $\varphi(f') = \varphi(f) - \varphi_f(v_i) = ((n-1)\pm2).180°$. and by induction we are done.

This leaves the case where all angles are either 90° or 270°. Since $n > 4$ and $\varphi(f) = (n\pm2).180°$ not all the angles can be equal. Hence there must be a k such that $\varphi_f(v_k) \neq \varphi_f(v_{k+1})$. Hence we have $\lambda(v_{k-1}v_kv_{k+1}v_{k+2}) = XYX$. Apply edge contraction to obtain $f' = v_1 \cdots v_{k-1}vv_{k+2} \cdots v_nv_1$. The edge contraction removed 360° from the angle sum and added 180°. Hence $\varphi(f') = ((n-1)\pm2).180°$. •

Definition 4.4: A *complement* of a path P *with respect to a square* σ is any path P^c in the graph such that PP^c is a cycle which simplifies to $\lambda(PP^c) = \sigma$.

Lemma 4.6: Given a path P, all its complements with respect to a square σ, which have the same start and end labels, have a *unique* simplified form.
Proof: Let $\lambda(P) = \alpha = X_1X_2 \cdots X_k$. Since α is a spiral we have $X_i = X_j$ for $i = j(4)$. Assume that $k > 4$ and that the spiral α and the square σ are either both clockwise or both counterclockwise. Then σ must be a substring of α. Since σ is a cyclic string we can assume that $\sigma = X_1X_2X_3X_4$.

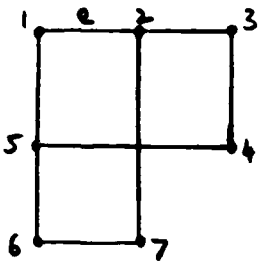
Let P^c be a complement of P with respect to σ and let $\overline{\lambda(P^c)} = \beta$. Since $k > 4$, β must spiral in the opposite direction to α. Since both α and β are simplified, $\alpha\beta$ can be simplified only at the borders between the two strings. Write $\beta = \beta_1\beta_2\beta_3$, such that $\overline{\beta_3\alpha\beta_1} = \alpha$ We are allowed to shift β_3 because $\alpha\beta$ is a cyclic string. Then it is clear that $\beta_1 \in \{X_{k-3}X_k, X_k, \varepsilon\}$ and $\beta_3 \in \{\varepsilon, X_1, X_1X_4\}$. β_2 is the 'essential part' of β. Since $|\alpha| = k$ and $\sigma = 4$, we must have $|\beta_2| = k-4$. From the possible values of β_1 and β_3, and the fact that β is a spiral opposite in direction to α , we can conclude that $\beta_2 = X_{k-1}X_{k-2} \cdots X_4$. We used $k > 4$ in order for β_2 not to be an empty string. Therefore $\beta = \{\varepsilon, X_k, X_{k-3}X_k\}X_{k-1} \cdots X_4\{\varepsilon, X_1, X_1X_4\}$, which is unique but for the start and end labels. The arguments in the cases where α and σ are in opposite directions and for $k \leq 4$ are similar. •

5. Biconnected Rectilinear Graphs

In this section we discuss an algorithm for recognizing biconnected rectilinear graphs. Note that the ordering relation λ induces a cyclic ordering of the edges incident at each vertex v. For convenience we will need the following definition.

Definition 5.1: Let v be a vertex in a rectilinear graph G. Define $L_G(v)$ to be the cyclic list of the neighbors of v in G in the counterclockwise order.

Using these lists, we can define the essential notion of a candidate face of a biconnected rectilinear graph.



$CF_1(e) = 2,1,5,4,3,2$ and $CF_2(e) = 1,2,7,6,5,4,3,2$

Figure 5.1

Candidate faces

Definition 5.2: Let $G = (V,E,\lambda)$ be a biconnected rectilinear graph. With each edge $e = (v_1,v_2)$, $\{v_1 > v_2\}$† we associate two lists of vertices called *candidate faces* $CF_1(e)$ and $CF_2(e)$ which are defined as follows. $CF_1(e) = v_1,v_2,\cdots,v_k,v_{k+1}$ where $v_i \neq v_j$ for $i,j \neq v_{k+1}$, $i \neq j$ and $v_{k+1} = v_i$ for some i, $1 \leq i < k-1$, such that for each l, $1 < l < k+1$, v_{l+1} is the successor of v_{l-1} in the cyclic list $L_G(v_l)$. $CF_2(e)$ is similarly defined but starting with v_2,v_1.

It is easy to see that CF_1 and CF_2 are uniquely defined. An illustration of this definition is given in figure 5.1.

We now need a lemma about biconnected undirected graphs. Let us define a biconnected graph to be *minimal* if for every edge e in the graph $G-e$ is not biconnected. The following lemma is taken from [2] and is stated without proof.

Lemma 5.1: If G is a *minimal* biconnected graph having at least four vertices then G contains a vertex of degree two.

Lemma 5.2: In any biconnected graph G which is not a simple cycle, there is a simple path $P = (v_1,v_2),(v_2,v_3),\cdots,(v_{r-1},v_r)$, $r \geq 2$, with the intermediate vertices (if any) v_i, $i \neq 1,r$ all having degree 2, such that the graph $G' = G-P$ is biconnected.

Proof: Transform the given graph G to another graph G'' by replacing all paths of the form $P = (v_1,v_2),(v_2,v_3),\cdots,(v_{r-1},v_r)$ where the vertices v_i, $i \neq 1,r$ all have degree 2, by the edge (v_1,v_r). So for each edge e in G'' we have a corresponding path P_e in G. Note that the degree of any vertex in G'' is at least three. If G'' has multiple edges between some two vertices, say u and w, then in G there must be at least two parallel paths between u and w. Since G is not a simple cycle any one of those paths will serve our purpose. If G'' does not have multiple edges then it must have at least 4 vertices. By lemma 5.1 G'' cannot be minimal. Therefore there is an edge e in G'' such that $G''-e$ is biconnected.

† For convenience we assume that V is a set of integers.

which implies that $G-P_e$ is also biconnected. ▪

The following theorem gives a necessary and sufficient condition for a biconnected rectilinear graph to be embeddable.

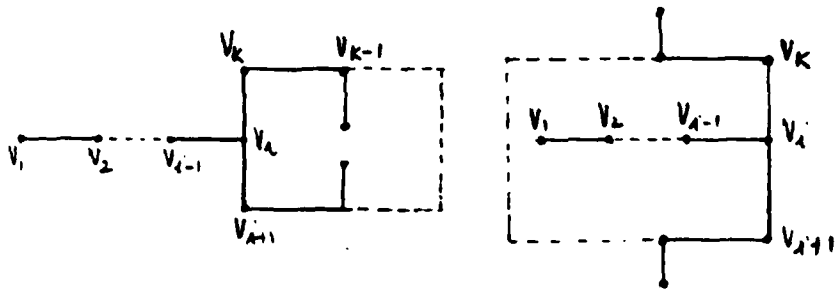


Figure 5.2
Two possible embeddings of $CF_1(e)$

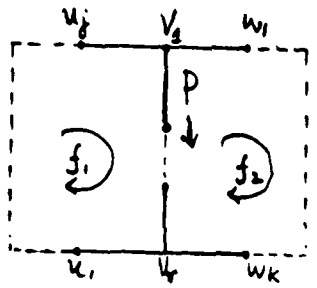


Figure 5.3
The two cycles f_1, f_2 and the path P

Theorem 5.1: Let $G = (V,E,\lambda)$ be an biconnected rectilinear graph with at least three edges. Then G is embeddable if and only if for each edge e in the graph both the candidate faces $CF_1(e)$ and $CF_2(e)$ represent simple embeddable cycles in the graph (i.e. the starting and ending vertices are identical, and it simplifies to a square). Moreover, if the graph is embeddable each such distinct candidate face corresponds to a face in the planar embedding.

Proof:

only if: Supposing for some edge $e=(v_1,v_2)$, $CF_1(e)$ is not a cycle, i.e $CF_1(e) = v_1,v_2,\cdots,v_k,v_{k+1}$ with $v_i = v_{k+1}$ for some i, $1 < i < k-1$. Suppose G is embeddable. Look at the cycle $v_i,\cdots,v_k,v_{k+1}$ in the embedding. Suppose that the edge (v_{i-1},v_i) is inside this cycle. There can be no other edges (u,v_j), $i \le j \le k$ ***inside*** this cycle, otherwise u would have appeared instead of v_{j+1} in $CF_1(e)$. From the planarity of the embedding, there can be no path from v_{i-1} to v_i other than the edge (v_{i-1},v_i). This contradicts the biconnectedness of G. The case where (v_{i-1},v_i) is ***outside*** the cycle is similar (both cases are depicted in figure 5.2).

Suppose $CF_1(e)$ is a cycle but is not embeddable. Since $CF_1(e)$ is a subgraph of G, G itself cannot be embeddable. Similar arguments holds for $CF_2(e)$.

If: The proof of this part is by induction on the number of edges. The basis for the induction are simple embeddable cycles, for which the claim is true by lemma 4.4. Assume that the claim is true for any biconnected rectilinear graph which has less than k edges. Let G be a biconnected rectilinear graph which is not a simple cycle and which has k edges. By lemma 5.2, there is a simple path $P = (v_1,v_2),(v_2,v_3),\cdots,(v_{r-1},v_r)$ with the vertices v_i, $i \neq 1,r$ all having degree 2, such that the graph $G' = G-P$ is biconnected. v_1 and v_r will have degree greater than two. Also assume that $v_1 > v_2$ and $e_{12} = (v_1,v_2)$.

Since all our candidate faces are cycles, if an edge e lies on a candidate face f then either $CF_1(e) = f$ or $CF_2(e) = f$. So each edge will be present in exactly two of these candidate faces. Hence the path P will appear in $CF_1(e_{12})$ and its reverse path will appear in $CF_2(e_{12})$. Let

$f_1 = CF_1(e_{12}) = v_1,v_2,\cdots,v_r,u_1,\cdots,u_j,v_1$,

$f_2 = CF_2(e_{12}) = v_r,v_{r-1},\cdots,v_1,w_1,\cdots,w_k,v_r$, and

$f_3 = v_1,w_1,\cdots,w_k,v_r,u_1,\cdots,u_j,v_1$.

It follows from the definition of the candidate faces f_1 and f_2 that the vertices u_j,v_2,w_1 appear consecutively in that order in $L_G(v_1)$ and that w_k,v_{r-1},u_1 appear similarly in $L_G(v_r)$ (see figure 5.3). Therefore for each edge in f_1 or f_2 which is not in P, the new candidate face in G' will be f_3 which is a simple cycle.

We still have to show that f_3 is embeddable. Since f_1 and f_2 are both embeddable $\varphi(f_1) = (r+j\pm2).180°$ and $\varphi(f_2) = (r+k\pm2).180°$. However, since f_1 and f_2 share the edge e_{12} it is implied by the definition of candidate faces that $\varphi(f_1) = (r+j+2).180°$ and $\varphi(f_2) = (r+k+2).180°$ is impossible. With a little bit of algebraic manipulation we can show that $\varphi(f_3) = ((j+k+2)\pm2).180°$. Since f_3 has $j+k+2$ vertices by lemma 4.5, it is embeddable. Thus the candidate faces for G' are the same as those for G, excepting for f_3 replacing the two faces f_1 and f_2. So for each edge in G' its two candidate faces are again simple embeddable cycles. By induction hypothesis G' is embeddable and each distinct candidate face corresponds to a face in its embedding. The orderings of the edges at the vertices v_1 and u_r imply that the end edges (v_1,v_2) and (v_{r-1},v_r) of the path P are both trying to go inside the face corresponding to f_3.

We are left to show that we can add the path P back without destroying embeddability. Find any rectilinear path P' in the face corresponding to f_3 in the embedding of G', that starts and ends with $\lambda((v_1,v_2))$ and $\lambda((v_{r-1},v_r))$ respectively. This is clearly possible although we may have to extend the grid in order for P' to lie on the grid lines. P' creates a face in the embedding with the path $P_1 = v_r u_1 u_2 \cdots u_j v_1$. If f_3 is not the outside face then $\overline{\lambda(P_1P')} = \overline{\lambda(f_1)} = \overline{\lambda(P_1P)} = \sigma$. The case when f_3 is the outside face is slightly more complicated. There are two such different paths P' depending on the new outer face that is created. However, for one of the two the above holds and suppose this is the one we chose. By definition both P and P' are complements of

P_1 with respect to σ, they also share the same start and end labels, and by lemma 4.6 we have $\overline{\lambda(P)} = \overline{\lambda(P')}$. Therefore $G'+P'$ can be obtained from G by applying a sequence of the four topological actions, and since $G'+P'$ is embeddable, by lemma 4.1 G is also embeddable. It is easy to see that the two new faces we get after inserting P in the embedding of G' correspond to f_1 and f_2. ∎

The above theorem leads to the following algorithm for recognizing embeddable biconnected rectilinear graphs. The algorithm also outputs the faces of the embedding if the graph happens to be embeddable.

```
Algorithm check-biconnected(G);
begin
    if G is an edge then return;
    if |E| > 3|V|−6 then
    begin
        write( 'not embeddable');
        quit
    end;
    for each edge e do
    begin
        mark[e,1]:= false;
        mark[e,2]:= false
    end;
    for each edge e do
        for i:= 1 to 2 do
        begin
            if not mark[e,i] then
            begin
                f := candidate-face( e, i );
                if not embed−cycle (f ) then
                begin
                    write( 'not embeddable' );
                    quit
                end;
                for each edge e' = (v1,v2) in f do
                    if v1 > v2 then mark[e',1]:= true
                    else mark[e',2]:= true;
                output (f )
            end
        end
end
```

Boolean function *embed-cycle(f)* returns value *true* if f is an embeddable cycle. If f is a cycle then we simplify using the reduction rules and check if we end up with a square. This can be done in time linear in the size of f. Function

call *candidate-face(e,i)* returns the candidate face $CF_i(e)$ and the function can be implemented exactly as described in definition 5.2. In the calls to this function, each edge e can be traversed at most twice, due to the flags mark[e,1] and mark[e,2]. Therefore the algorithm runs in time $O(|V|)$. We conclude this section with a lemma which will let us identify the outer face in a rectilinear graph.

Lemma 5.3: Let G be an embeddable biconnected rectilinear graph. For all interior faces f in the embedding of G, $\varphi(f) = (n-2).180°$, and for the unique exterior face f_e, $\varphi(f_e) = (n+2).180°$.

Proof: Consider the embedding of G. The faces of the embedding are determined by G, and are simple polygons in the plane. By the definition of φ, for every interior we count the interior angles, and for the exterior face we count the exterior angles. The lemma follows. (Remember that if G is a simple cycle, the embedding has two faces). •

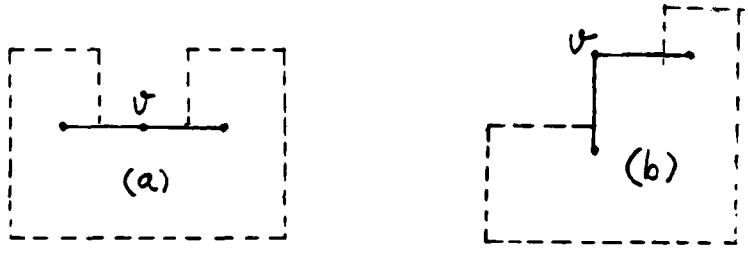


Figure 5.4
Shapes **U** and **W**

Lemma 5.4: Let G be an embeddable biconnected rectilinear graph, f_e the exterior face in its embedding and v a vertex on f_e. If $\varphi_{f_e}(v) = 180°$ then G can be embedded inside a polygon of shape **U**, as shown in figure 5.4a. If $\varphi_{f_e} = 270°$ then G can be embedded inside a polygon of shape **W**, as shown in figure 5.4b.

Proof: Easy and left to the reader. •

6. Articulation Vertices

In this section we examine the conditions under which the embeddability of the biconnected components of the graph implies the embeddability of the graph itself. Clearly, this will depend on the way components meet at articulation vertices. In figure 3.2, we showed two examples of nonembeddable rectilinear graphs, each of which decomposes into two embeddable biconnected rectilinear graphs.

In those cases, the two biconnected components are not "compatible" at the articulation vertex. However, the situation need not be so local. Figure 6.1

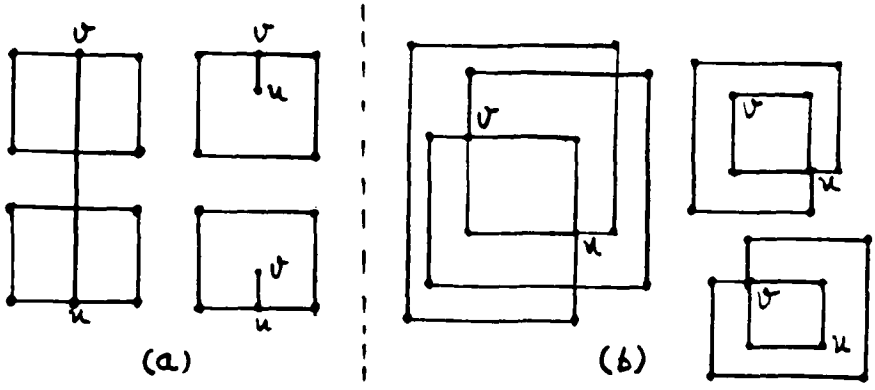


Figure 6.1
Decompositions of nonembeddable graphs

depicts two nonembeddable graphs, each of which decomposes into three embeddable biconnected components, so that the components meeting at each articulation vertex are compatible. Note that an edge is a (trivial) biconnected component.

If v is an articulation vertex in a graph G, then its removal results in several connected subgraphs G_i of G. We will refer to the subgraphs G_i+v, as the subgraphs *meeting* at v. Throughout this section we will implicitly assume that we are dealing with rectilinear graphs whose biconnected components are embeddable.

Definition 6.1: Let B_1 and B_2 be two nontrivial biconnected components of a rectilinear graph G that share an articulation vertex v. Then B_1 and B_2 are said to *interlace* if the horizontal edges at v belong to B_1 and the vertical edges belong to B_2. We also say that v is an *interlace* vertex. Any articulation vertex that does not have this property is said to be *interlace-free*.

Lemma 6.1: A rectilinear graph G which has an interlace articulation vertex v is not embeddable.
Proof: Let B_1 and B_2 be the two biconnected components sharing the vertex v. Since B_1 and B_2 are nontrivial, the horizontal edges at v lie on a cycle in B_1 and the vertical edges lie on a cycle in B_2. It is impossible to draw G on the plane without these two cycles crossing. •

Definition 6.2: Let B_1 and B_2 be two non interlacing biconnected components of G that share an articulation vertex v, and assume B_1 is nontrivial. Then B_2 is said to be *inside* B_1 (or B_1 *dominates* B_2 at v) if either (i) v is not on the exterior face of B_1, or (ii) edges (v,u) and (v,w) at the vertex v are on the exterior face of B_1, and u,w are consecutive in that order in $L_G(v)$ (note that they are always consecutive in $L_{B_1}(v)$). If neither B_1 dominates B_2 nor B_2 dominates B_1, then B_1 and B_2 are said to be *outside* each other.

The intuition behind the above definition is that in the embedding, one biconnected component must lie wholly inside some face of the other if one edge of it does. This is due to the planarity criterion. Clearly, if biconnected components B_1 and B_2 that share an articulation vertex v dominate each other, the graph is not embeddable (this is the case in figure 3.2b).

Let B_1 and B_2 be two biconnected components of a graph G that share an articulation vertex v, such that B_1 dominates B_2. Let G' be the subgraph of G meeting at v, that contains B_2. If G is embeddable then in any embedding of G, all of G' should lie inside one face of B_1. This suggests extending the relation "dominate" as follows:

Definition 6.3: Let $\mathbf{B} = \{B_1, B_2, \cdots, B_m\}$ be the set of biconnected components of G. We say that B_i ***dominates*** B_j if there exists a biconnected component B_k and an articulation vertex v, such that (*i*) B_k and B_i share v, (*ii*) B_i dominates B_k at v, and (*iii*) B_j and B_k are both subgraphs of the same connected subgraph meeting at v.

Let us denote by $V(G)$ the vertex set of the graph G and by $E(G)$ the edge set.

Lemma 6.2: If in a rectilinear graph G, there exists some pair of biconnected components B_1 and B_2 that dominate each other, then G is not embeddable.
Proof: If B_1 and B_2 share an articulation vertex v, then as mentioned earlier G is not embeddable. Suppose that B_1 and B_2 are disjoint. Since B_1 and B_2 dominate each other, there must be articulation vertices v_1, v_2, biconnected components B_1', B_2', and subgraphs G_1, G_2, such that for $i = 1,2$, (*i*) B_i and B_i' share v_i, (*ii*) B_i dominates B_i' at v_i, and (*iii*) G_i is one of the subgraphs meeting at v_i and contains B_i'. Let us assume that G is embeddable. From (*i*) $v_2 \in V(G_1)$, (*ii*) G_1 lies wholly inside B_1 in the embedding, and (*iii*) $V(G_1) \cap V(B_1) = \{v_1\}$, we can conclude that v_2 must be properly inside a polygon defined by the face f_1 of B_1 containing v_1. Similarly v_1 should be properly inside the polygon defined by a face f_2 of B_2 containing v_2. Therefore some vertices of f_2 must lie outside f_1 and the two faces must intersect, and hence G is not embeddable. •

Given a rectilinear graph G, with set of biconnected components $\mathbf{B}$ and set of articulation vertices $\mathbf{A}$, we can construct a tree T of biconnected components such that $V(T) = \mathbf{A} \cup \mathbf{B}$, and $E(T) = \{(v, B) \mid v \in \mathbf{A}, B \in \mathbf{B}, v \in V(B)\}$.

Lemma 6.3: Let G be a rectilinear graph with the set of biconnected components $\mathbf{B}$ and tree of biconnected components T. Let B be a leaf in the tree T which is adjacent to an articulation vertex v of degree 2 in T. If B dominates B' the other biconnected component adjacent to v in T, then B dominates every other biconnected component in $\mathbf{B}$.
Proof: The only two subgraphs meeting at v are B and $G - B + v$ and the proof follows from definition 6.3. •

If no two biconnected components dominate each other, then the relation "dominate" induces a partial order on **B**. A nondominating element in this partial order is a biconnected component which does not dominate any biconnected component.

Corollary 6.1: If for a rectilinear graph G, "dominate" is a partial order, then there exists a nondominating biconnected component which is a leaf in the tree T of biconnected components.

Proof: Any trivial biconnected component (which is just an edge) must be nondominating. If any vertex in T (corresponding to an articulation vertex in G) is adjacent to two leaves, then either the two leaves are nontrivial and not dominating, or one of them is a trivial biconnected component. If no vertex in T is adjacent to two leaves, then all leaves are adjacent to vertices of degree 2, and there are at least two such leaves. If two of these leaves are dominating, then by lemma 6.3 the two leaves dominate each other which is a contradiction that "dominate" is a partial order. In fact all of these leaves must be nondominating. ▪

Theorem 6.1: Let G be a rectilinear graph and **B** its set of biconnected components. G is embeddable if and only if

(i) every biconnected component B in **B** is embeddable,

(ii) every articulation vertex in G is interlace-free, and

(iii) "dominate" induces a partial order on **B**.

Proof: The necessary part follows from lemma 6.1 and lemma 6.2.

The sufficient part is shown by induction on the number of vertices. The basis for induction is any biconnected rectilinear graph. Let G be not biconnected with $|V(G)| = n$. Assume that the claim is true for all smaller graphs. Look at the tree T of biconnected components. By corollary 6.1, there exists a leaf B in T which is nondominating. Let v be the articulation vertex shared by B and $G' = G-B+v$, the rest of the graph. G' being a subgraph of G also satisfies the conditions of the claim. By induction hypothesis G' is embeddable. By condition (i), B is also embeddable. If B is a single edge it is easy to add the edge to the embedding of G'. Assume B is nontrivial. Since B is nondominating, v must lie on the exterior face f_e of B and $\varphi_{f_e}(v) \neq 90°$ (why?).

Embed G' and B separately and consider the vertex v in both embeddings. If $\varphi_{f_e}(v) = 180°$, then v is on only one edge in G'. Add · new grid lines to the embedding of G', create the shape **U** as shown in figure 6.2a, magnify the embedding, and embed B in the **U** as in lemma 5.4a. If $\varphi_{f_e}(v) = 270°$, then v is either on just one edge in G' , or on two perpendicular edges in G'. In both cases, add new grid lines, create the shape **W** and embed B as shown in figure 6.2b. ▪

Before we describe an algorithm for testing embeddability, we need an algorithm for testing whether "dominate" is a partial order on the set of biconnected components. From the tree T of biconnected components, we construct $\bar{T}$ a

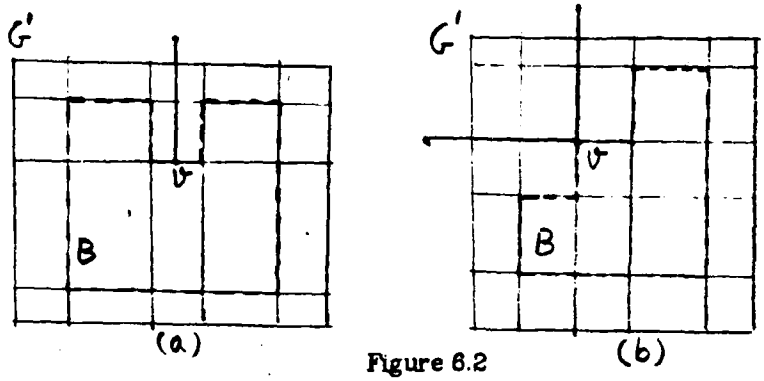


Figure 6.2
Adding B to the embedding of G'

partially directed tree as follows. Assume that no biconnected component dominates and is dominated at the same vertex. If so then "dominate" is not a partial order. Direct edge (v,B) from B to v if B dominates at v. Direct edge (v,B) from v to B if B is dominated at v. Leave all the other edges undirected.

This partially directed tree $\overline{T}$ can be constructed in linear time as follows. Find the faces of each of the biconnected components using the algorithm *check-biconnected*. This takes $O(|V|)$ time. Check for dominations at each articulation vertex as described in definition 6.2. There are at most 4 biconnected components at each articulation vertex and hence there are at most 12 (ordered) pairs to be tested for domination (in fact only 2 tests are necessary, how?). Construct $\overline{T}$ by directing the edges of T as described earlier. Note that articulation vertices and biconnected components can be found in $O(|V|)$ [1]. For each vertex x in $\overline{T}$, denote by $d_{in}(x)$, $d_{out}(x)$, and $d(x)$, the number of incoming arcs, the number of outgoing arcs, and the number of undirected edges respectively. The rest of the algorithm is given below.

```
Algorithm check-dominate-po(G);
begin
    construct T̄;
    for each vertex x in T̄ do
        if d_in(x) > 1 then
        begin
            write ( 'not a partial order' );
            quit
        end.
    if search (T̄) then write ( 'yes, partial order' )
    else write ( 'not a partial order' )
end.

function search(T̄): boolean;
begin
    if T = φ then search := true
```

else begin
if $\exists B \in \mathbf{B}$ with $d_{out}(B)=0$, $d_{in}(B)+d(B)=1$ **then**
begin
Let v be the neighbor of B;
if $d_{in}(v)+d_{out}(v)+d(v)=1$ **then** *search* := *search*$(\bar{T}-\{B\})$
else *search* := *search*$(\bar{T}-\{B,v\})$
end else *search* := *false*
end

The above algorithm can be easily shown to be correct using definition 6.3 and corollary 6.1. The boolean function *search* can be implemented nonrecursively to run in linear time by maintaining a queue of the leaves of $\bar{T}$.

Given the biconnected components and articulation vertices, checking that the articulation vertices are interlace-free can be done in $O(|V|)$ time. Let *check-interlace-free* be a procedure that checks a given articulation vertex for interlace-freedom. We end this section with a $O(|V|)$ algorithm for testing embeddability of rectilinear graphs.

Algorithm *check-rectilinear(G)*;
begin
Decompose G into its biconnected components;
for each biconnected component B **do** *check-biconnected(B)*;
for each articulation vertex v **do** *check-interlace-free(v)*;
check-dominate-po(G)
end

7. An Embedding Algorithm

In the previous section we gave an algorithm for testing embeddability This algorithm can be easily modified into an algorithm which gives an embedding. However, the complexity of this naive algorithm would be $O(|V|^3)$. The reasoning is as follows. The path P' that we find in the proof of theorem 5.1 could be $O(|V|)$ long. For each topological action that we apply on this path to transform it to the path P, we update the coordinates of the vertices in the embedding once. Thus for each path added we require $O(|V|^2)$ time. There can be $O(|V|)$ such paths and hence the complexity of the algorithm is $O(|V|^3)$. To reduce the complexity to $O(|V|)^2)$, we have to make sure that the path P' is never longer (asymptotically) than the path P. In this case the sum of the lengths of all such paths P' is $O(|V|)$, and the $O(|V|^2)$ complexity follows. In the following, we show how we can always find such paths, describe the algorithm, and analyze its complexity.

Lemma 7.1: Let G be a planar biconnected multigraph with minimum degree

three. Then any embedding of G has an interior face of size at most five.

Proof: The dual G^d of G is also a planar graph. Since G has minimum degree 3, G^d is a simple graph. Hence G^d has at least two vertices of degree ≤ 5 [2]. G is biconnected and hence one of the vertices must correspond to a face of size ≤ 5. •

Lemma 7.2: Given an embedding of a planar biconnected graph G, which is not a cycle, there is a simple path P, such that (*i*) the interior vertices of P all have degree 2, (*ii*) the end vertices of P have degree ≥ 2, (*iii*) P appears in an interior face f in the planar embedding, and (*iv*) $5.|P| \geq |f|$.

Proof: As in the proof of lemma 5.2, transform G to G' by replacing all paths with property (*i*) and (*ii*) by edges. By lemma 7.1, G' has an interior face f of size at most 5. The longest of all the paths in G corresponding to the edges of f will satisfy conditions (*iii*) and (*iv*). •

To get an embedding of a given rectilinear graph, we first test if the graph is embeddable and then apply the following algorithm.

Algorithm *embed-rectilinear(G)*;
begin
  for each biconnected component B **do** *embed-biconnected (B)*;
  join-the-embeddings;
end

Algorithm *embed-biconnected(B)*;
begin
  get-long-path (P, P_1, σ);
  embed-biconnected ($B - P$);
  find-path-in-embedding (P', P_1, σ);
  apply-actions-and-transform (P', P)
end

Procedure *get-long-path* returns paths P,P_1, and square σ, such that P satisfies the conditions of lemma 7.2, and the interior face $f = PP_1$ simplifies to σ. By lemma 7.2 such a path exists.

Procedure *find-path-in-embedding* traces a path P' in the embedding of $B-P$, such that P' starts and ends in the same directions as P, and $P'P_1$ simplifies to σ. P' and P are both complements of P_1 with respect to the square σ. Since PP_1 is an interior face, P' can be obtained by starting in the required direction, then following the path P_1 in the embedding of $B-P$, and ending in the required direction (figure 7.1). This will result in P' being a complement of P_1 with respect to σ. We have $|P'| = O(|P_1|) = O(|P|)$.

Procedure *apply-actions-and-transform* applies a sequence of the four topological actions to P' in the embedding of $B-P+P'$ and transforms it to P thus resulting in a embedding of B. This is done by first simplifying the path P'

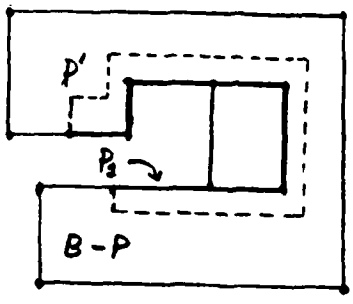


Figure 7.1
Finding the path P' in the embedding of $B-P$

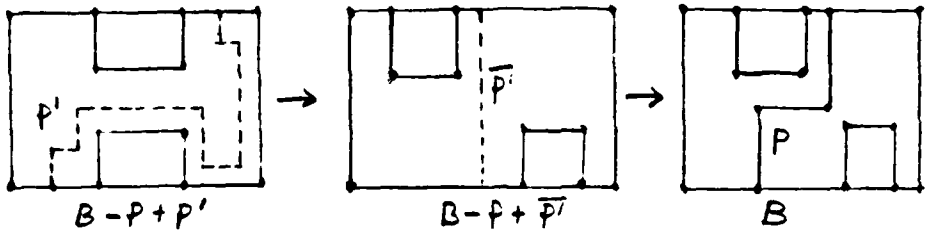


Figure 7.2
Path addition, simplification and expansion

and then expanding the simplified path to get the path P (figure 7.2). The number of actions applied will be $O(|P|+|P'|) = O(|P|)$.

Procedure ***join-embeddings*** takes the embeddings of the biconnected components and puts them together to get an embedding for G. This is done essentially following the proof of theorem 6.1. Find a nondominating component B. Recursively embed $G' = G - B$. Join the embeddings of B and G' using the shapes U or W as shown in figure 6.2

The algorithm can be shown to be correct using the material developed in the previous three sections. We now analyze the complexity of each step in the algorithm and show that the total complexity is $O(|V|^2)$.

Procedure ***join-embeddings*** updates each coordinate at most once per recursive call. The total number of calls is bounded by the number of biconnected components. Hence this procedure takes $O(|V|^2)$ time.

Procedure ***get-long-path*** can be implemented to run in $O(|V|)$ time each time it is called. Remember that we can get the faces of a biconnected graph from the testing algorithm, and searching all faces to get the required face takes linear time. Procedure ***find-path-in-embedding*** takes $O(|P_1|) = O(|V|)$ time. These two procedures will be invoked at most $O(|V|)$ time. Hence total

time spent in these calls is $O(|V|^2)$.

Procedure ***apply-actions-and-transform*** applies a sequence of $O(|P|)$ actions. Each edge in G will appear in only one such path P. Hence the sum of the lengths of all such paths P is $O(|V|)$. Each action updates at most $O(|V|)$ coordinates. Therefore the time spent in calls to this procedure is $O(|V|^2)$.

8. Consistent Rectilinear Graphs

Certain rectilinear graphs cannot be drawn on the grid even if we relax the planarity criterion. We say that a rectilinear graph $G(V,E,\lambda)$ is ***consistent*** if it can be drawn on the grid satisfying the ordering relation λ. In other words, G is consistent if the set of equality and inequality constraints generated in part 1 of definition 2.2 is consistent.

The equality constraints define an equivalence relation on the set of coordinates of the vertices of G. Let us denote by $e(x)$ the equivalence class containing the coordinate x. Denote by I_x and I_y the sets of x-coordinate and y-coordinate inequality constraints respectively. Construct two directed graphs $G_x(V_x,E_x)$ and $G_y(V_y,E_y)$ as follows:

$$V_x = \{e(x) \mid x = x(a), a \in V\} \text{ and } E_x = \{(x_1,x_2) \mid x_1 > x_2 \in I_x\}$$

V_y and E_y are similarly defined.

It can be easily shown that G is consistent if and only if the two directed graphs G_x and G_y are both acyclic. A solution to the coordinates which satisfies the constraints will correspond to a nonplanar embedding of G on the grid. This can be obtained by performing the topological sort operation [4] on the two acyclic digraphs. In fact this will yield a solution that minimizes the area of the rectangle bounding the embedding.

In a nonplanar embedding of a consistent rectilinear graph on the grid, all crossings are between horizontal edges and vertical edges. The vertical edges can be assigned one layer, and the horizontal edges can be assigned a second layer. In other words the 'thickness' [2] of a consistent rectilinear graph is less than or equal to two. A generalization of the rectilinear graph embedding is the problem of embedding a rectilinear graph with layers preassigned, in which no two edges belonging to two different layers cross. This remains an open problem.

9. Extensions and Open Problems

As mentioned in the previous section, the embedding problem for layer assigned rectilinear graphs is still to be solved. This has important applications in VLSI. It is easy to show that if a rectilinear graph is allowed to be disconnected, then the optimal area embedding problem is NP-complete (reduction from two dimensional bin packing). However, the question is open for connected rectilinear graphs.

On a more theoretical side, we can define 'triangular', 'hexagonal' and other polygonal graphs, and consider the embedding problems on appropriate grids. However, it seems that the work in this paper does not generalize easily. This is mainly because these polygonal graphs lack the nice simplification properties of rectilinear graphs. Geometry seems to dominate over topology in these polygonal graphs.

We conclude with a note that our $O(|V|^2)$ embedding algorithm will be implemented in ALI. In ALI, layouts are described in a hierarchical fashion, and hence the algorithm can be applied hierarchically on a cell by cell basis. In this case the complexity of obtaining the embedding is the sum of the quadratic complexity over all cell instances.

10. Acknowledgements

This problem was originally raised by Professors Bob Sedgewick and Dick Lipton. We wish to thank Professors Dick Lipton and Jacobo Valdes for several useful discussions. The first author's research was supported by DARPA under grant# N0014-82-K-0549. The second author's research was supported by an IBM fellowship.

11. References

[1] Aho, A. V., Hopcroft, J. E., Ullman, J. D., ***The Design and Analysis of Computer Algorithms***, Addison-Wesley, 1974.

[2] Behzad, M., Chartrand, G., Lesniak-Foster, L., ***Graphs & Digraphs***, Wadsworth International Group, 1981.

[3] Hopcroft, J. E., Tarjan, R. E., "Efficient Planarity Testing", ***J. Assoc. Comput. Mach.***, 21, (549-568), 1974.

[4] Knuth, D. E., ***The Art of Computer Programming***, vol. 1, ***Fundamental Algorithms***, Addison-Wesley, 1971.

[5] Leiserson, C. E., "Area Efficient Graph Embeddings (for VLSI)", ***Proc. 21st Symp. on the Foundations of Computer Science***, October 1980.

[6] Lipton, R. J., Sedgewick, R., Valdes, J., "Programming Aspects of VLSI", ***Proc. 9th Annual ACM Symp. on Principles of Programming Languages***, 1982.

[7] Lipton, R J., North, S. C., Sedgewick, R., Valdes, J., Vijayan, G., "ALI: a Procedural Language to Describe VLSI Layouts", ***Proc. of the 19th Design Automation Conference***, Las Vegas, June 1982.

[8] Sethi, R., "Testing for the Church-Rosser property", ***J. ACM***, 21, 4, Oct. 1974.

[9] Vijayan, G., "Completeness of VLSI layouts", VLSI Memo# 1, Department of Electrical Engineering and Computer Science, Princeton University.

[10] Vijayan, G., Wigderson, A., "Planarity of Edge Ordered Graphs", (in preparation).

[11] Wigderson, A., "The Complexity of the Hamiltonian Circuit Problem for Maximal Planar Graphs", Technical Report #298, Department of Electrical Engineering and Computer Science, Princeton University, February 1982.

On Driving Many Long Lines in a VLSI Layout

Vijaya Ramachandran

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

Abstract We assume that long wires represent large capacitive loads, and investigate the effect on the area of a VLSI layout when drivers are introduced along many long wires in the layout. We present a layout for which the introduction of drivers along long wires *squares* the area of the layout; we show, however, that the increase in area is never greater than this, if the driver can be laid out in a square region. We also show an area-time trade-off for a single long wire by which we can reduce the area of its driver to $\Theta(l^q)$, $q<1$, from $\Theta(l)$, if we can tolerate a delay of $\Theta(l^{1-q})$ rather than $\Theta(\log l)$; and we obtain tight bounds on the worst-case area increase in general layouts having these drivers, using the Brouwer fixed-point theorem. We also derive results for the case when drivers are embedded in rectangles that are not square. Finally, we extend the use of our upper-bound technique to other layout problems.

1. Introduction

The presence of long wires in a VLSI layout slows down the performance of the circuit due to signal propagation delays. This effect can be minimized by using a driver at the head of each long wire. However, the drivers themselves occupy space, and the question arises whether drivers can be introduced efficiently when there are many long wires in a layout. For example, consider the simple case of a long bidirectional wire. A driver needs to be introduced at each end of the wire, and the presence of each driver increases the length along which the other driver has to drive the signal. A large network with many long wires may contain several instances of mutual interactions of this and other kinds, and it is not clear whether drivers can be introduced in an area-efficient manner under these conditions (or even whether they can be introduced at all). This is the problem we analyze in this paper.

The question of delays along long wires is a complex one, and no consensus has been reached as yet on the correct way of modeling this. Some papers (e.g., [BK80], [V80]) assume that a constant propagation delay models the current situation quite well. But it is generally accepted that this is not a good approximation of the physical situation. Mead and Conway [MC80] and Chazelle and Monier [CM81] suggest that a wire is basically a distributed RC-network, and the delay thus is proportional to the square of its length. This delay can be reduced to a linear delay by using repeaters along the wire. Linear delay is also the asymptotic limit imposed by the speed of light. Thompson [T80, T81] suggests that the resistance of a long wire can be ignored and that the wire can be viewed as a large capacitive load whose capacitance is proportional to its length. This view is also presented in [MR79] and is supported by Bilardi, Pracchi, and Preparata [BPP81] who solved the diffusion equation for a long wire and suggest that, for predicted future gains in technology, the wire can, indeed, be modeled as a purely capacitive load. For such a load, it is well-known that the delay can be reduced to the logarithm of the capacitance by introducing a sequence of drivers which occupies area proportional to the capacitance [MC80].

In this paper, we assume that the wire can be modeled as a purely capacitive load. We address the following question that arises naturally in this context, but has not been examined before: *What is the effect on the area of a layout when drivers are introduced to speed up signals along many long wires?* We justify our capacitive-model assumption by appealing to the simulation results of [BPP81]. Further, if a long wire must, indeed, be modeled as an RC-network, then the introduction of repeaters to reduce the delay will increase the area only by a constant factor. This is because such repeaters are of constant size, and a long wire may thus be modeled as a sequence of short wires connected by nodes of fixed size. In the capacitive model, on the other hand, the area occupied by a driver increases with the length of the wire it drives, and thus, drivers *cannot* be abstracted as constant-size nodes. In fact, we show that there can be definite area penalties when there are many drivers in a layout. Our results are of particular significance for upper-bounds in area-time products for VLSI layouts, since most previously derived bounds have ignored the delay along long wires. Either linear delay should be assumed along connecting wires (this would represent either "RC" wires with repeaters, or "capacitive" wires without drivers), or the area expansion caused by drivers for capacitive wires should be taken into consideration; alternatively, some intermediate design for drivers can be used from the spectrum of designs we suggest in section 4. But upper-bounds derived using constant delay along all wires, and no area expansion for drivers do not model the physical situation well.

For a wire of length l, the most familiar type of driver occupies $\Theta(l)$ area while reducing the delay from $\Theta(l)$ to $\Theta(\log l)$. In this paper, we present a layout for which the introduction of such drivers along long wires *squares* the area of the layout; we show, however, that the increase in area is never greater than this, if the driver can be laid out in a square region. We prove the upper-bound by a new technique that uses a fixed-point theorem; we believe the proof technique is important in its own right. We also show an area-time trade-off for a single long wire by which we can reduce the area of its driver to $\Theta(l^q)$, $q<1$, from $\Theta(l)$, if we can tolerate a delay

This research was supported in part by NSF under Grant MCS-8203063, and in part by DARPA under ONR N00014-82-K-0549.

of $\Theta(l^{1-\epsilon})$ rather than $\Theta(\log l)$; and we obtain bounds on the worst-case area increase in general layouts having these drivers, again using the Brouwer fixed-point theorem. We also examine the case when drivers cannot be laid out efficiently in square regions.

The paper is organized as follows: Section 2 presents the VLSI model we use. In section 3 we present a graph on n^2 vertices with $\Theta(n)$ long wires for which the layout area expands from $\Theta(n^2)$ to $\Theta(n^4)$ after introducing drivers for the long wires. Section 4 examines the area-time trade-off for drivers, and sections 5 and 6 use the Brouwer fixed-point theorem to find general tight upper-bounds for the area expansion caused by drivers in arbitrary layouts. Section 7 generalizes the technique to other layout problems, and section 8 concludes the paper with a review and some open questions.

2. The Model

We assume a layout model similar to the one used by Thompson[T80] and Leiserson[L80]. The VLSI circuit is abstracted as a graph having bounded vertex degree and it is embedded on a planar grid subject to the following constraints:

1) Each vertex represents a processing element and occupies a constant area. Distinct vertices of the graph are embedded at distinct grid intersection points.

2) Edges have unit width and are routed along grid lines with the restriction that no two edges touch one another except possibly when crossing perpendicular to each other. Also, an edge cannot be routed over a vertex it does not connect. Each edge represents a connecting wire. We will refer to an edge as a *wire* or a *line*.

We assume that we are given such a layout for a circuit with certain edges identified as long wires. Note that the number and positions of long wires is layout-dependent, and the same circuit may have another layout with shorter lines. We shall not go into the question of derigning layouts to minimize the presence of long lines. Some work has been done on minimizing the length of the longest line in a layout for certain classes of graphs ([PRS81], [RS81], [BL82]), but, as we mention in section 6, the minimization criterion in our case is a different one, which has not been studied so far. We will, therefore, assume that the layout is given, and we will introduce drivers where needed by making *local expansions* without distorting the layout configuration.

A long wire is a bidirectional element electrically, but in a VLSI circuit, it usually connects active unidirectional devices, so that the signal always originates at one fixed end of the wire (which we will call the *head* of the wire) and propagates towards the other end. For the rest of this paper, we assume that wires come with the direction of signal flow specified, and that bidirectional wires are decomposed (conceptually) as two edges connecting the same vertices, but having opposite signal flow.

As mentioned in the introduction, we assume that the line is purely capacitive, and the delay thus grows linearly with l, the length of the line. This delay can be reduced by using drivers to speed up the signal. Many different types of driver designs are possible, and they differ in the signal speed-up they offer, and the area they occupy. We analyze these time-space trade-offs in section 4. For the present, we assume the most familiar type of driver, which reduces the delay to $\Theta(\log l)$ at the expense of occupying $\Theta(l)$ area. We also assume that the driver can be laid out in a square of side $k\sqrt{l}$, for some

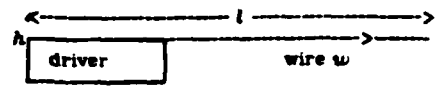


figure 1
A driver introduced along
part of a long wire

constant k. This is not strictly true, since the driver has a transistor whose channel is $\Theta(l)$ wide, and hence it is more naturally laid out in a long, narrow rectangle. Clever design methods, however, such as the use of a zig-zag poly line for the gate in MOS technology, can be used to overcome this problem. We have more to say on this point in section 6.

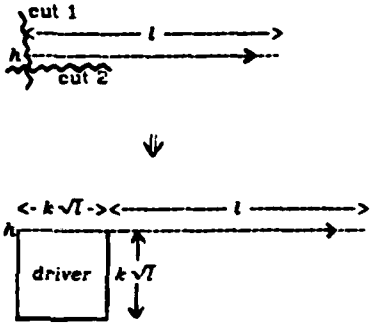


figure 2
Introducing a driver by making
cuts in a layout

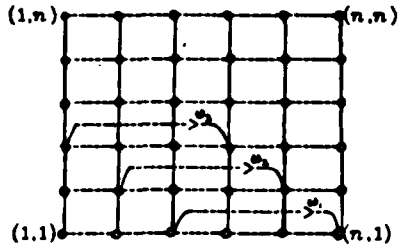
Given a long line w of length l in a layout, an area-efficient way to achieve signal speed-up would be to introduce a long skinny driver (the natural design) along a portion of w (see figure 1). However, VLSI layouts are dense in general, and the above scheme will not work if there are vertical lines present in this region. In such a case, we make two "cuts" or "slices" (a notion introduced by Leiserson [L80]) at the head h of w, each $k\sqrt{l}$ wide, one each in the vertical and horizontal directions (figure 2). Any edge that is broken by the cut is then joined across the cut by a straight line. This forms a square region of $k\sqrt{l}$ side at h with no edges or vertices from the original layout. We introduce the driver in this region. This construction does not disturb the layout configuration, and introduces the driver by local expansions. It increases the length of each side of the layout by the size of the cut. *It also increases by $k\sqrt{l}$, the length of each horizontal wire in the layout which intersects the vertical line drawn through h and each vertical wire that intersects the horizontal line drawn through h.* These other wires will now require larger drivers than the ones they would have needed in the absence of a driver at h. The introduction of drivers at these other lines will, in turn, affect the lengths of more lines, and could even increase the length of w so that the area of its driver would have to be revised upward.

We are now in a position to state the problem we are going to analyze:

Given a VLSI layout with certain edges specified as long wires, what is the increase in the area of the layout in the worst case when drivers are used to speed up signals along these long wires?

3. A Worst-case Example

Consider the graph layout of figure 3. It consists of a square mesh on n^2 vertices (assume n even), together with $n/2$ long horizontal lines $w_1, w_2, \cdots, w_{n/2}$. If we assume the vertices to be laid out in an $n \times n$ grid (there will be a slight expansion needed to accommodate the long wires along grid lines, but we can ignore this for "order-of-magnitude" arguments) and let $m = n/2$, then w_1 runs from vertex $(m,1)$ to vertex $(n,1)$, w_2 runs from $(m-1\ 2)$ to $(n-1,2)$, $\cdots$, and w_m runs from $(1,m)$ to $(m+1,m)$. The signal goes from left to right along each of these of these lines.



$n=6$

figure 3
A graph layout with $\Theta(n^2)$ area and $\Theta(n)$ long lines of length $\Theta(n)$ each

We consider now the increase in the area of the layout when we introduce drivers on wires w_1 to w_m. Notice that the presence of the underlying square-mesh structure forces us to make cuts to introduce the drivers. Since $n/2$ drivers are introduced diagonally, and the length of each wire is $\Theta(n)$ to start with, the length of each side of the layout increases to at least $\Theta(n \cdot n^{1/2})$, so that the area of the layout increases from $\Theta(n^2)$ to at least $\Theta(n^3)$. However, this does not account for the increase in the lengths of some wires due to the introduction of drivers on other wires. We now derive a nonlinear recurrence relation that accounts for this effect, and use it to bound the increase in area as $\Theta(n^4)$.

Since the signal travels from left to right along each of the long wires, the length of w_1 is not affected by the introduction of drivers at the heads of the other long wires. In general, the length of w_i increases by exactly the lengths of the drivers introduced on w_1 to w_{i-1}. If we let d_i be the width of each side of the driver for wire w_i, then we have

$$d_1 = km^{1/2}$$
$$d_2 = k(m+d_1)^{1/2}$$
$$\vdots$$
$$d_i = k\left(m+\sum_{j=1}^{i-1} d_j\right)^{1/2}$$

Let s_i be a new variable defined by

$$s_1 = m + d_1$$
$$s_2 = m + d_1 + d_2$$
$$\vdots$$
$$s_i = m + \sum_{j=1}^{i} d_j$$
$$= s_{i-1} + d_i$$
$$= s_{i-1} + k\left(m + \sum_{j=1}^{i-1} d_j\right)^{1/2}$$
$$= s_{i-1} + k\sqrt{s_{i-1}}$$

For convenience, define $s_0 = m$. Note that $s_i + i$ gives the width of the layout from the rightmost end to the left end of the ith driver, and $s_m + m$ gives the length of each side of the layout after the introduction of drivers on all of the long wires. We find bounds for this value below:

Lemma: There exist constants k_1 and k_2 such that for any integer $m > 0$,
$k_2 i^2 + m \geq s_i \geq k_1 i^2 + m$ for $i \geq \sqrt{m}$.

Proof We prove both sides of the inequality by induction.

Assume $s_i \geq k_1 i^2 + m$
Since $s_0 = m$, the assumption holds for s_0 for all values of k_1. Assume the result holds up to s_i. Then,

$$\begin{aligned} s_{i+1} &= s_i + k\sqrt{s_i} \\ &\geq k_1 i^2 + m + k(k_1 i^2 + m)^{1/2} \\ &> k_1 i^2 + m + k k_1^{1/2} i \\ &= k_1(i+1)^2 + m + (k k_1^{1/2} - 2k_1) i - k_1 \end{aligned}$$

Hence, $s_{i+1} \geq m + k_1(i+1)^2$ for $k_1 < (k^2/4)$ and for sufficiently large i. For $k_1 \leq (k^2/9)$, the result holds for all values of i. So we have $s_i \geq (k^2/9) i^2 + m$ for all $i \geq 0$.

Assume $s_i \leq k_2 i^2 + m$
Assume the result holds up to s_i. Then,

$$\begin{aligned} s_{i+1} &= s_i + k\sqrt{s_i} \\ &\leq k_2 i^2 + m + k(k_2 i^2 + m)^{1/2} \\ &\leq k_2 i^2 + m + k((k_2+1) i^2)^{1/2} \text{ for } i \geq m^{1/2} \\ &= k_2(i+1)^2 + m - (2k_2 - k(k_2+1)^{1/2}) i - k_2 \\ &< k_2(i+1)^2 + m \text{ for } k_2 \geq (k^2/8)(1+\sqrt{1+16/k^2}). \end{aligned}$$

Hence we have $s_i \leq (k^2/8)(1+\sqrt{1+16/k^2}) i^2 + m$ for $i \geq \sqrt{m}$. For $i = \sqrt{m}$, we need $s_{\sqrt{m}} \leq k_2 m + m$, and since $s_{\sqrt{m}} < m + k\sqrt{m}\sqrt{m + 2k(1+k)m}$, the inequality holds when $k_2 \geq k\sqrt{1 + 2k + 2k^2}$.

Therefore, $s_i \leq k_2 i^2 + m$ for all $i \geq \sqrt{m}$ when $k_2 \geq \max\left((k^2/8)(\sqrt{1+16/k^2}),\ k\sqrt{1+2k+2k^2}\right)$.

The required result follows from combining the two inequalities.∎

Thus, the width of the layout after the introduction of drivers, which is given by $s_m + m$, is $\Theta(m^2) = \Theta(n^2)$ and the area thus grows to exactly $\Theta(n^4)$, the square of the initial area. The width of the driver for w_i is given by:

$$\begin{aligned} d_i &= k\left(m + \sum_{j=1}^{i-1} d_j\right)^{1/2} \\ &= k\sqrt{s_i} \\ &= \Theta(i), \text{ for } i \geq \sqrt{m}. \end{aligned}$$

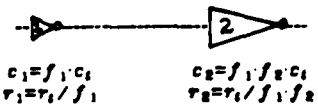
Hence there are $\Theta(n)$ drivers of width $\Theta(n)$ in this layout. Note also that many of the short wires have their

lengths greatly increased by the introduction of drivers for wires w_1 to w_m. For instance, the wire connecting the vertex $(1,n)$ to the vertex $(2,n)$ has its length increased from 1 grid unit to $\Theta(n)$ grid units. We may now want to introduce drivers for these wires, too, but since we a deriving a lower bound in this section, we will ignore this point.

The layout of figure 3 is simple in the sense that it has no mutual interaction: there do not exist wires w_i and w_j such that the driver of w_i increases the length of w_j and the driver of w_j increases the length of w_i. In general, we could have a more complex pattern of long wires in which such mutual interactions occur in many ways. However, in section 6, we present the surprising result that the penalty in area never becomes worse than the square of the original area.

4. Area-Delay Trade-off for a Driver of a Long Wire

When a large capacitive load c_L is driven by an inverter with resistance r_i, the delay is the product $r_i c_L$ (note that r_i here refers to an "average" resistance, since the resistance and the delay vary depending on whether the input is low or high. In what follows, we make some other simplifications of the physical situation, but these will affect the constant factor only, so that the "order-of-magnitude" results hold). If the resistance and capacitance of a minimum-size inverter are r_i and c_i respectively, then the basic unit of delay is $t=r_i c_i$ and this represents the delay when a minimum-size inverter drives the gate of another minimum-size inverter. In terms of this unit t, a minimum-size inverter driving a load capacitance c_L causes a delay of $c_L t / c_i$, and this grows linearly with the load capacitance. When the load is a long wire, this delay grows linearly with l, the length of the wire, assuming the width and thickness of the wire remain fixed.



channel width of inv. $1 = f_1$ (minimum channel width)
channel width of inv. $2 = f_2$ (channel width of inv. 1)

figure 4
Delay on one stage of a driver sequence

To reduce the delay to more acceptable levels, we use a sequence of drivers with successively wider channels to drive the load capacitance. To understand why this scheme works, consider the delay when one inverter with channel width f_1 times the minimum-size inverter drives the gate of another inverter whose channel is f_2 times wider than that of the driving inverter (see figure 4). The resistance of the first inverter is r_i / f_1 and the capacitance of the second inverter is $f_1 f_2 c_i$ and hence the delay of this stage is $f_2 t$. If we now have n inverter stages, starting with a minimum-size inverter as the first stage, a driver with a very wide channel as the nth stage to drive c_L, and intermediate stages with successively wider channels, then the total delay is the sum of the delay through the n stages. Let f_i be the ratio of the channel width of the $i+1$st inverter to that of the ith inverter for $i=1,2,\cdots,n-1$, and let f_n be the ratio of c_L to the capacitance c_n of the last driver (figure 5).

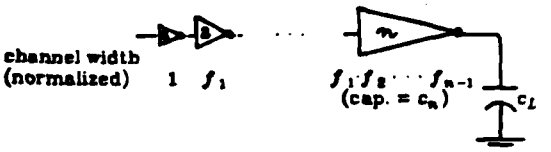


figure 5
A sequence of drivers for load c_L

We have $f_1 f_2 \cdots f_n = c_L / c_i$ and the total delay through the n stages is $T=(f_1+f_2+\cdots+f_n)t$. If we fix each f_i to be equal to a constant f, then the delay $T=\log_f(c_L/c_i)$, and this is the minimum delay we can obtain up to a constant (see [MC80]). However, the area occupied by the drivers in this case grows linearly with c_L, as we show below, and it may be desirable to reduce this area penalty at the expense of a slightly greater delay. We show below that such a space-delay trade-off exists. Some of these results are implicit in the work of Lin and Linholm [LL75] and Jaeger[J75].

Let the area occupied by a minimum-size inverter be A_0. Then, an inverter whose channel is f times as wide as a minimum-size inverter requires an area of about fA_0. Hence the total area A_t occupied by the sequence of drivers with logarithmic delay is given by

$$\begin{aligned} A_t &= A_0(1+f+f^2+\cdots+f^{n-1}) \\ &= \Theta(f^n) \\ &= \Theta(c_L). \end{aligned}$$

Now, consider the general sequence of drivers described above. The area occupied in this case is

$$A_g = A_0(1+f_1+f_1f_2+\cdots+f_1f_2\cdots f_{n-1}).$$

Assume that we can tolerate a delay of $\Theta(l^q), 0<q<1$ $(=\Theta((c_L)^q))$. Since the area of the last driver dominates the above expression for area, we look for the smallest size for the last driver which will still result in a delay of not more than $\Theta(l^q)$. Since $c_n=c_L/f_n$, and the delay through the last stage is $f_n t$, the best we can do is to set $f_n=\Theta(l^q)$, and this results in an area of $\Theta(l^{1-q})$ for c_n. Now, the question is whether we can accommodate the other drivers to occupy not more than $\Theta(l^{1-q})$ area and cause no greater than $\Theta(l^q)$ delay. The answer to this is 'yes', and in fact, we can accomplish this in many ways. We describe one method of doing this, which minimizes the constant on the delay term. For this, we make c_n the load so that we need to drive a load of $\Theta(l^{1-q})$. We know from the design for drivers with logarithmic delay, that this can be done with $\Theta((1-q)\log l)$ delay and area $\Theta(l^{1-q})$ using $\Theta((\log c_n/c_i))$ stages. Hence the total area for the n stages remains $\Theta(l^{1-q})$ while the delay is incremented by only a $\Theta(\log l)$ term and hence remains $\Theta(l^q)$.

We have thus shown the following interesting area-delay trade-off for the driver of a single long wire of length l: *For any q, $0<q<1$, we can design a driver with delay $\Theta(l^q)$ and area-delay product $\Theta(l)$. If, however, we need to reduce the delay to $\Theta(\log l)$, which is the minimum delay achievable, then the area-delay product goes up to $\Theta(l \cdot \log l)$.*

5. An Upper-Bound Proof Technique Using Fixed-Points

Let $F(x_1,x_2,\cdots,x_m)$ be a mapping from $\mathbf{R}^m$ to $\mathbf{R}^m$ defined by

$$F(x_1,\cdots,x_m)=(f_1(x_1,\cdots,x_m),\cdots,f_m(x_1,\cdots,x_m)),$$

where each f_i is a mapping from $\mathbf{R}^m$ to $\mathbf{R}$. Then, a *fixed-point* of F is a vector $(x_1,\cdots,x_m)\in\mathbf{R}^m$ such that

$$F(x_1,\cdots,x_m)=(x_1,\cdots,x_m), \text{ i.e.,}$$
$$f_i(x_1,\cdots,x_m)=x_i, \text{ for } i=1,\cdots,m.$$

The following are some well-known results from real analysis (see e.g., Bartle [B64]).

Brouwer Fixed-Point Theorem Let $B>0$, and let D be the hypersphere defined by $D=\{x\in\mathbf{R}^m:|x|\le B\}$. Then, any continuous function with domain D and range contained in D has a fixed-point.

Fact 1 Any real continuous function on a bounded closed set in $\mathbf{R}^m$ has a maximum and a minimum.

Fact 2 For $B>0$, the hypersphere defined by $D=\{x\in\mathbf{R}^m:|x|\le B\}$ is a closed bounded set.

We will need the following two definitions:

Definition 1 Let $f(x_1,\cdots,x_m)$ be a continuous mapping from $\mathbf{R}^m$ to $\mathbf{R}$ and let $g(x)$ be the mapping from $\mathbf{R}^+$ to $\mathbf{R}^+$ such that for each $R\ge 0$, $g(R)$ gives the maximum absolute value of f in the closed hypersphere of radius $R\sqrt{m}$. We call g the *behavior* function of f. Note that by facts 1 and 2, g always exists.

Definition 2 A function $g(x)$ from $\mathbf{R}^+$ to $\mathbf{R}^+$ is *sublinear* in x if there exists an $r\ge 0$ such that, for all $x\ge r$, we have $g(x)\le x$. We call r a *breakpoint* of the function g.

We now prove

Theorem 1: Let F be a continuous mapping from $\mathbf{R}^m$ to $\mathbf{R}^m$ given by
$F(x_1,\cdots,x_m)=(f_1(x_1,\cdots,x_m),\cdots,f_m(x_1,\cdots,x_m))$,
and for each f_i, let the behavior function g_i be sublinear in x, with breakpoint r_i. Then,
1) the function F has a fixed-point within a hypersphere of radius $R\sqrt{m}$, where $R=\max_i r_i$, and
2) each component of the fixed-point is no greater than R in magnitude.

Proof To prove 1), it is sufficient, from Brouwer's theorem, to prove that the range of F is contained on or within the prescribed hypersphere when the domain is similarly constrained. For this, it is sufficient to prove that, for each i, $g_i(R)\le R$, since this will guarantee that the Euclidean length of the range is no greater than $\sqrt{m}R$. But this follows immediately from definition 2, since the functions g_i are sublinear with breakpoints r_i, and for each i, $r_i\le R$.

Note that, by requiring $g_i(R)\le R$ for each i, we are constraining the range to lie within a closed hypercube inscribed within the hypersphere of the domain. This additional restriction ensures that each component of the fixed-point is at most R in magnitude, and this proves 2), and the theorem.•

In our application to layout problems, and in particular, to the problem of drivers, we will be interested in a special class of functions $f_i(x_1,\cdots,x_m):\mathbf{R}^m\to\mathbf{R}$ defined by

$$f_i(x_1,\cdots,x_m)=k(l_i+\sum_{j=1}^{m}\delta_{ij}x_j)^p, \text{ when } (l_i+\sum_{j=1}^{m}\delta_{ij}x_j)\ge 0,$$
$$=0, \text{ otherwise.} \qquad (1)$$

where k and l_i are positive constants, $0<p<1$, and $0\le\delta_{ij}\le K$, for some positive constant K. Usually, δ_{ij} is either 0 or 1. For such functions, we have the following corollary to Theorem 1.

Corollary 1.1: Let F be a mapping from $\mathbf{R}^m$ to $\mathbf{R}^m$ given by
$F(x_1,\cdots,x_m)=(f_1(x_1,\cdots,x_m),\cdots,f_m(x_1,\cdots,x_m))$,
where the f_i's are given by (1). Then F has a fixed-point, each component of which is positive and has magnitude no greater than R, where $R=\max(k_1(l_1)^p,\cdots,k_m(l_m)^p,k_{m+1}m^{p/(1-p)})$, the k_i's being positive constants independent of m.

Proof It is easy to prove that each f_i has behavior function $g_i(x)=f_i(x,\cdots,x)$. Let $r_i=$(a solution of the equation $k(l_i+\sum_{j=1}^{m}\delta_{ij}x)^p=x$). Then, since $p<1$, g_i is sublinear with breakpoint r_i. Hence from Theorem 1, each component of a fixed-point of F has magnitude no more than R, where $R=\max_i r_i$.

Case 1 Assume that each $l_i\le(\sum_{j=1}^{m}\delta_{ij}r_i)$. Then we require $c_i(\sum_{j=1}^{m}\delta_{ij})^p(r_i)^p=r_i$, for some constant $c_i\le 2k$, or, $r_i=c'_i(\sum_{j=1}^{m}\delta_{ij})^{p/(1-p)}$, where $c'_i=(c_i)^{1/(1-p)}$. The maximum possible value for each δ_{ij} is K. Hence, each coordinate of the fixed-point is guaranteed to have a value no greater than $R=k_{m+1}m^{p/(1-p)}$, for some constant k_{m+1} (that depends on k and K).

Case 2 There exist l_k's such that each $l_k>(\sum_{j=1}^{m}\delta_{kj}r_k)$. Let l_i be the maximum value of such l_k's. Then $\sum_{j=1}^{m}\delta_{ij}r_i=O(l_i)$, and hence we need $r_i=k_i l_i^p$ for some constant $k_i<2k$. Hence, in this case, each coordinate of the fixed-point is guaranteed to have a value at most $R=\max(k_i l_i^p,k_{m+1}m^{p/(1-p)})$.

By combining the two cases, the required result follows. Clearly, since the range of F lies entirely in the positive orthant, each component of the fixed-point must be positive.•

Corollary 1.2: If at most r of the δ_{ij} can be nonzero for each function f_i in (1), then each component of a fixed-point of F is positive and has magnitude no greater than R, where $R=\max(k_1 l_1^p,\cdots,k_m l_m^p,k_{m+1}r^{p/(1-p)})$.

Proof Immediate from the proof of Corollary 1.1.•

6. General Upper Bounds for Area Penalty in Layouts with Drivers

We initially assume that a driver for a wire of length l can be embedded in a square of side kl^p ($p\le 1/2$). When $p=1/2$, this corresponds to the standard driver with logarithmic delay. Since we are now proving upper bounds, we will assume all wires in the layout to be long. This may be a necessary assumption, since a wire that

was short initially may become long due to the introduction of drivers on other wires (as we saw in section 3).

Now, assume that the layout is in an $n \times n'$ grid ($n \geq n'$). Hence the area of the layout before introducing drivers is $A = nn'$. We note the following points about long lines and their drivers in this layout:

1) If each long line is l units long, then there are at most $2A/l$ long lines in the layout.

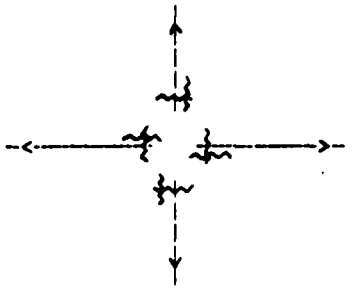


figure 6
The position of a cut for a driver

2) The cut perpendicular to the direction of the wire is always made with the position of the head fixed, and on the side the wire extends (figure 6). Thus, *if there are* two wires with their heads along the same vertical (or horizontal) line and with opposite signal flow, their drivers do not overlap the same horizontal (or vertical) region. Since this simplification at most doubles the effect of drivers at any single vertical or horizontal position, the "order-of-magnitude" results remain unaffected. The cut in the direction parallel to the wire is made on the side of the wire which lies on a clockwise rotation from the wire (see again figure 6). With this convention, we can introduce drivers on all four lines at a vertex, if needed.

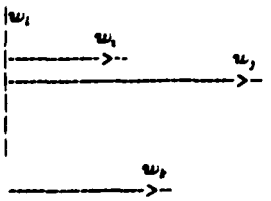
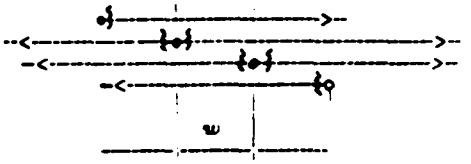


figure 7
The effect of long wires with their with heads aligned

3) Consider the situation in figure 7. All of the lines w_i, w_j, w_k, w_l have their heads along the vertical line through w_i, and the signal flow on each wire is such that a cut is required immediately to the right of w_i to accommodate the driver for each wire. In such a case, the width of the widest cut is sufficient to include all drivers. A similar result holds for horizontal cuts. Note that this means that we need make at most two cuts at each vertical and horizontal grid position (the factor of two arises from the convention we adopted in observation 2 above, regarding the position of the cut). Thus, at most $2(n+n')$ cuts will be needed to accommodate drivers in any $n \times n'$ layout. The *widths* of the cuts need to be determined, and we bound these values later in this section to obtain tight upper-bounds on the worst-case area expansion caused by the introduction of drivers.

4) If a wire bends (i.e., goes both vertical and horizontal), the analysis still holds. We assume, however, that a wire bends at most a constant number of times, and hence a wire can have length no more than Kn, for some constant K.

5) If a wire is l grid units long, then its length is affected by at most $2l$ other drivers. This worst case happens when there are the heads of two long wires (either vertical or horizontal) at each of the $l-1$ inner grid positions of w, and the signal flow on these wires is directed so that cuts are required on both sides of the grid position; and there is a long wire at each end grid position of w, and each of these wires requires a cut on the side affecting the length of wire w. This situation is shown in figure 8.



length of wire w = 3 units

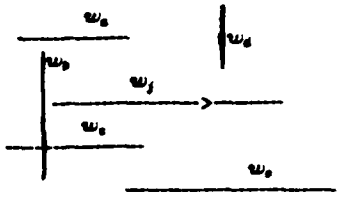
figure 8
Worst-case effect of other drivers on the length of a wire w initially k grid units long

Consider a general layout with m long wires $w_1, w_2, \cdots, w_m$ of initial lengths $l_1, l_2, \cdots, l_m$. As mentioned earlier, bidirectional wires are treated as two separate wires, with opposite signal flow. Let the driver for the ith wire be embedded in a square of side d_i. We need to solve for the d_i's, $i = 1, 2, \cdots, m$. For this, we note that the length of the ith wire, l_i, is increased to a new value l_i' by the presence of drivers on some of the other wires, i.e., we have

$$l_i' = l_i + \sum_{j=1}^{m} \delta_{ij} d_j, \quad i = 1, \cdots, m,$$

where δ_{ij} is nonzero if the driver of wire w_j increases the length of w_i, and is 0 otherwise. Since we allow a wire to bend a constant number of times, a single cut can expand a wire by up to K' times the width of the cut, and hence δ_{ij} is a nonnegative integer having value no more than K. Now, the driver of w_j will increase the lengths of all horizontal wires that intersect the vertical line through the head of w_j, and all vertical wires that inter-

sect the horizontal line through the head of w_j (see figure 9).



$\delta_{aj}=1, \delta_{bj}=1, \delta_{cj}=1, \delta_{dj}=0, \delta_{ej}=0.$
The signal flow can be in either direction in each of wires w_a to w_e.

figure 9
Determining δ_{ij}

Hence a simple preprocessing of the layout gives the values for $\delta_{ij}, i,j=1,\cdots,m$. Note that we can treat both horizontal and vertical wires uniformly because of our assumption that the driver occupies a square region. Later, we will need to modify our equations when we consider drivers embedded in rectangles that are not square. The value for each d_i is given by

$$d_i = k(l_i')^p = k(l_i + \sum_{j=1}^{m} \delta_{ij} d_j)^p. \qquad (2)$$

Let us rewrite equation (2) by defining d_i in a functional form, and extending its domain to $\mathbb{R}^m$ as follows:

$$d_i = f_i(d_1,\cdots,d_m) = k(l_i + \sum_{j=1}^{m} \delta_{ij} d_j)^p, \quad i=1,\cdots,m, \text{ when } (l_i + \sum_{j=1}^{m} \delta_{ij} d_j) > 0,$$
$$= 0 \quad \text{otherwise.}$$

Let $F(d_1,\cdots,d_m)$ be the mapping from $\mathbb{R}^m$ to $\mathbb{R}^m$ defined by

$$F(d_1,\cdots,d_m) = (f_1(d_1,\cdots,d_m),\cdots,f_m(d_1,\cdots,d_m)).$$

Then the solutions of the system of equations (2) correspond exactly to the fixed points of the function F, i.e., tc the values of $d_1,\cdots,d_m$ such that

$$F(d_1,\cdots,d_m) = (d_1,\cdots,d_m).$$

From the results of the previous section, we can now immediately derive upper bounds on the width of each driver. We first determine this for the case when the layout is in a square $n \times n$ grid. We later show that this is the worst case.

Assume that each l_i is l units long, and hence, from observation 5, up to $2l$ of the δ_{ij} can be nonzero for each i. Hence, by Corollary 1.2, each component of the fixed-point has magnitude no greater than R, where

$$R = \max(k_1 l^p,\cdots,k_m l^p, k_{m+1}(2l)^{p/(1-p)}) = O(l^{p/(1-p)}).$$

Hence the width of any driver in this layout will never become greater than $cl^{p/(1-p)}$ for some constant c. The maximum width occurs in the case when $l=\Theta(n)$ (since this is the maximum value for l, by observation 4), and this width is bounded by $cn^{p/(1-p)}$. The worst-case over p occurs when $p=1/2$ and in this case the width is $O(n)$. But we know from the example of section 3 that a driver can indeed become as wide as $\Theta(n)$. Hence this bound is tight. There can be up to $\Theta(n)$ wires in the layout (by observation 1), and hence we may have to make $\Theta(n)$ cuts in each of the two directions, so that the layout area will increase from $A=n^2$ to $\Theta(n^4)=\Theta(A^2)$. This upper bound in the increase in area is again matched by the lower bound of section 3. For $p<1/2$, the area increases to $O((n^{1/(1-p)})^2)=O(A^{1/(1-p)})$.

If $l=\Theta(n^r), 0<r<1$, then the width of each driver is $O(n^{rp/(1-p)})$, but by observation 1, there can be up to $\Theta(n^{2-r})$ such lines in the layout. We thus have more than $\Theta(n)$ drivers in this case; however, by observation 3, if we make two cuts of width $\Theta(n^r)$ at each vertical and horizontal grid position where drivers are needed (there are at most $4n$ such cuts), then we can accommodate all drivers. Each side of the layout will expand to at most $\Theta(n^{1+rp/(1-p)})$ and the area grows to $\Theta(n^{2+2rp/(1-p)})=\Theta(A^{1+rp/(1-p)})$. Since $r<1$, this increase is less than the increase for the case when $r=1$.

If we had a mixture of wires of varying length, the width expansion is still $O(n^{1/(1-p)})$ by the same argument. In the worst case, we may have to make a cut for a driver at each side of each vertical and horizontal grid position. There are $4n$ such vertical and horizontal grid positions, and the worst case occurs when $\Theta(n)$ cuts of width $\Theta(n^{p/(1-p)})$ are required in each of the two directions. We have thus proved the following theorem:

Theorem 2: If the driver of a long wire of length l can be embedded in a square region of side $\Theta(l^p), 0<p\le 1/2$, then the area A of any VLSI layout embedded on a square grid increases to at most $O(A^{1/(1-p)})$ with the introduction of such drivers along long wires.

We have already shown that this bound is tight when $p=1/2$. We now show that this holds for any $p\le 1/2$.

Lemma: If the driver of a long wire of length l can be embedded in a square region of side $\Theta(l^p), 0<p\le 1/2$, then there exists a VLSI layout whose area increases from A to $\Omega(A^{1/(1-p)})$ when such drivers are introduced along long wires.

Proof We use, once again, the layout of figure 3. The recurrence relation for s_i is now

$$s_i = s_{i-1} + k(s_{i-1})^p,$$

with $s_0=m=n/2$. As before, s_m+m gives the width of each side of the layout after the introduction of drivers. We prove by induction that $s_m \ge \Omega(m^{1/(1-p)})$.

Assume $s_i \ge c \cdot i^{1/(1-p)}+m$.
The result holds for s_0 for all values of c. Assume that the result holds up to s_{i-1}. Then,

$$\begin{aligned} s_i &= s_{i-1} + k(s_{i-1})^p \\ &\ge c(i-1)^{1/(1-p)} + m + k(c(i-1)^{1/(1-p)} + m)^p \\ &\ge c(i-1)^{1/(1-p)} + m + kc^p(i-1)^{p/(1-p)} \\ &= ci^{1/(1-p)}((1-1/i)^{1/(1-p)} \\ &\qquad + (k/(c^{1-p}i))(1-1/i)^{p/(1-p)}) + m \\ &= ci^{1/(1-p)}(1-1/i)^{p/(1-p)}(1+((k/c^{1-p})-1)/i) + m. \end{aligned}$$

Since $p \le 1/2$, we have $p/(1-p) \le 1$, and hence $(1-1/i)^{p/(1-p)} \ge (1-1/i)$. Hence,

$$s_i \ge c i^{1/(1-p)}(1-1/i)(1+((k/c^{1-p})-1)/i)+m \ge c i^{1/(1-p)}+m \text{ when } i \ge 2 \text{ and } c \le (k/3)^{1/(1-p)}.$$

We also need $s_1 = m + k m^p \ge c + m$. Hence, $s_i \ge c i^{1/(1-p)}+m$ for all i and for $c \le \min((k/3)^{1/(1-p)}, k m^p)$. Set $m=1$ in the min expression to obtain a value for c that is independent of m.

The width of each side of this layout after the introduction of drivers is $s_m + m = \Omega(m^{1/(1-p)}) = \Omega(n^{1/(1-p)})$, and the required result for the area increase follows.•

Next, we consider the case when the driver is embedded in a rectangle that is not square. In this case, we show below that, in the worst case, the longer side of the driver dominates the summation in equation (2) by proving the following theorem:

Theorem 3: Assume that the driver for wire w_i of length l_i can be embedded in a rectangle that has length $a_i = k_1 l_i^s$ along the direction of the wire and width $b_i = k_2 l_i^t$ in the perpendicular direction. If $0 < s, t < 1$, then the worst-case increase in the area A of a VLSI layout on a square grid is $\max(\Theta(A^{1/(1-s)}), \Theta(A^{1/(1-t)}))$ when such drivers are introduced at the heads of long wires.

Proof Since the driver is no longer symmetrical in the two directions, we must now modify the equations for the f_i's to:

$$a_i = f_{1i}(a_1,\ldots,a_m,b_1,\ldots,b_m) = k_1(l_i + \sum_{j=1}^{m}(\delta_{1ij}a_j + \delta_{2ij}b_j))^s$$
$$b_i = f_{2i}(a_1,\ldots,a_m,b_1,\ldots,b_m) = k_2(l_i + \sum_{j=1}^{m}(\delta_{1ij}a_j + \delta_{2ij}b_j))^t, \quad (3)$$

where the domains of f_{1i} and f_{2i} are extended to $\mathbb{R}^{2m}$ as in the previous case. The δ_{1ij}'s and the δ_{2ij}'s are determined as before (see figure 9), except that the first subscript of 1 refers to lines parallel to wire i, and that of 2 refers to lines perpendicular to it. Thus, in figure 9, we have the following values: $\delta_{1aj}=1$, $\delta_{2aj}=0$, $\delta_{1bj}=0$, $\delta_{2bj}=1$, $\delta_{1cj}=1$, $\delta_{2cj}=0$, and the other values are all zeros. We now require the fixed-point of the function

$$F(a_1,\ldots,a_m,b_1,\ldots,b_m) = (f_{11},f_{12},\ldots,f_{1m},f_{21},f_{22},\ldots,f_{2m}),$$

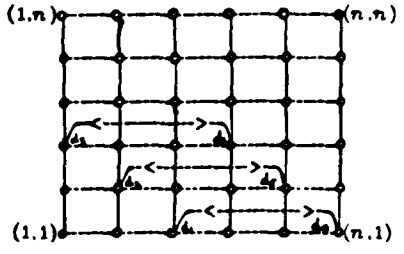
(where the arguments on each f_{ij} on the right hand side are $a_1,\ldots,a_m,b_1,\ldots,b_m$.) Again, if each long wire is $\Theta(n^r)$ long to start with, we obtain from Corollary 1.2, the following upper-bound for the width of each driver:

$$R = \max(\Theta(n^{rs/(1-s)}), \Theta(n^{rt/(1-t)})) = \Theta(n^{rp/(1-p)}), \text{ where } p = \max(s,t).$$

As before, the worst case occurs when $r=1$, and this gives us a maximum width of $O(n^{p/(1-p)})$ for each driver along its longer side. We still may need $\Theta(n)$ cuts of this width in each of the two directions to introduce the drivers, and this gives us the required result for the maximum increase in area.•

This bound is once again tight, and the lower bound can be proved using the layout of figure 3 with $n/2$ vertical long lines added on with their heads along a diagonal. Note that in this case, s or t may be greater than 1/2, and so the area blow-up will be more than the square of the original layout. In particular, if $p=1$, the above bound ceases to hold. (This will be the case for a long driver whose area is proportional to the length of the wire it drives, i.e., the natural design for a standard driver.) In this case, the convergence of the system of equations (3) for specific layouts depends on the constant k_1. If $k_1 \ge 1$, then, clearly, drivers cannot be introduced in any layout containing a long bidirectional wire. In fact, for any fixed value for k_1, we can design layouts in which drivers cannot be introduced, as we show in the following theorem.

Theorem 4: If the driver of a wire of length l must be embedded in a rectangle whose length is kl, for some constant k, then there exist layouts in which drivers cannot be introduced on all long wires.



$n=6$

figure 10
A layout to prove Theorem 4

Proof Consider the graph of figure 10. It is the same as the one in figure 3, except that all long wires are bidirectional. We will prove that, for $n \ge 2/k$, drivers cannot be introduced on all long lines in this layout. As before, let us write out the equation for each d_i. For instance,

$$d_1 = k(l_1 + d_{m+1} + d_{m+2} + \cdots + d_n)$$
$$d_2 = k(l_2 + d_1 + d_{m+2} + \cdots + d_n).$$

(where $m = (n/2)$), and the equations for the other d_i's can be written out by inspection. Since this is a system of linear equations, this can be rewritten in matrix form as

$$Ad = kl,$$

where A is an $n \times m$ matrix of the coefficients of the d_i's, l is a vector of the l_i's, and d is a vector of the d_i's. We need a solution of this system of equations in which the d_i nonnegative. For this, we note that the matrix A is an M-matrix when $n < (2/k)$, and from the theory of such matrices, it is simple to prove that the system of equations has a solution with all components of d nonnegative only when $n < 2/k$ [BP79].•

For particular layouts, we may still obtain convergence, but with a large area penalty. For example, the layout of section 3 (which has no mutual interaction between drivers and wires) will require area $\Omega((1+k)^m)$ after the introduction of such drivers. This result is easily verified from the recurrence relation for s_i which, in this case, is $s_i = (1+k)s_{i-1}$.

We pointed out in section 4 that the driver has a last stage with a very wide channel that dominates the area of the driver. Thus such a driver will, indeed, be very long, and have constant width. The introduction of such a driver in the circuit can be disastrous in terms of area blow-up (if not impossible). However, it should be possi-

ble to redesign the channel to occupy a more square region, possibly with some penalty in the total area required. Thus, if the driver originally required area $\Theta(l^s)$, then this design might result in a driver of length $\Theta(l^s)$ and width $\Theta(l^t)$ with $\Theta(l^{s+t})>\Theta(l^s)$, but with $0<s,t<1$. In such a case, drivers can *always* be introduced in any layout of long wires with a polynomial blow-up in area given by Theorem 3 (the degree of this polynomial will be very high if s or t is close to 1). In practice, we expect driver designs in square areas with no extra area penalty, and this results in the squaring of the area in the worst case.

Finally, we look into the case of a rectangular VLSI layout in an $n \times n'$ grid ($n \geq n'$). As before, if each long wire is l units long, then the width of any driver is $O(l^{p/(1-p)})$ by Corollary 1.2. By observation 1, the number of wires in the layout is $O(nn'/l)=O(A/l)$, where A is the area of the layout before the introduction of drivers. If $l=O(n')$, then we may have to make $2n$ cuts along the longer side, increasing its length to $O(n\, l^{p/(1-p)})$, and similarly, $2n'$ cuts along the shorter side, increasing its length to $O(n'\, l^{p/(1-p)})$. Hence the area increases from nn' to $O(nn'\cdot l^{2p/(1-p)})=O(A^{1/(1-p)})$. If $l=\Omega(n')$, then, since the number of wires in the layout is $O(nn'/l)$, the number of cuts along the longer side is also no greater than $O(nn'/l)$, so that the length of that side increases to $O((nn'/l)l^{p/(1-p)})$. The increase in length of the shorter side remains $O(n'\cdot l^{p/(1-p)})$. Hence the area increases to no more than $O((nn'/l)\, n'\cdot l^{2p/(1-p)})=O(A^{1/(1-p)})$, since $l=O(n)$ by observation 4. Hence the worst case area expansion for rectangular layouts due to the introduction of drivers is no greater than that for square ones.

To summarize, we have shown that

1) if the drivers described in section 4 can be embedded in a square region without extra area penalty, then in the worst case, the area of a VLSI layout grows quadratically with the introduction of drivers for long wires. This worst case occurs when drivers require area proportional to the length of the wire they drive (i.e., the standard design); when the layout is in an $n \times n$ square and has $\Theta(n)$ long wires of length $\Theta(n)$ each; and when the introduction of drivers requires $\Theta(n)$ cuts of width $\Theta(n)$ in the layout in each of the two directions.

2) if a driver requires a rectangular embedding, then the worst case increase in area is $\Theta(A^{1/(1-p)})$, where the longer side of a driver rectangle for a wire of length l is $\Theta(l^p)$ grid units long. If $p \geq 1$, then there exist layouts for which drivers cannot be introduced.

7. Application to Other Layout Problems

The technique developed in the previous two sections can be used in other related problems in VLSI. Consider the following question:

Given a VLSI layout, what is the global effect of making many local spot expansions that mutually interact?

The introduction of drivers for long lines is one example of this problem. Another example is fault-tolerant computing in VLSI. Von Neumann[vN56] has suggested a method of constructing reliable combinatorial circuitry using components that have a certain probability of failure. The construction requires replication of the basic components many times. This method does not carry over directly to VLSI circuits, since the connecting wires here are also likely to fail, and the probability of failure increases with the length of the wire. This probability can be reduced by increasing the width. It is generally accepted that a fault can occur anywhere on the surface of a silicon wafer with a constant probability, and thus, the occurrence of a fault can be modeled as a random variable with a Poisson distribution. Under this model, it can be shown that the probability of failure along a wire can be bounded by any constant $\varepsilon>0$ by making the width of the wire proportional to the log of the length. But if the width is increased after the circuit is laid out (and this is the only logical way, since the the lengths of wires in the layout cannot be known before the circuit is laid out), then this will result in increasing the lengths of other wires, and it is not clear what the global effect of many such local transformations is.

We analyze the general problem by assuming that, in the worst case, a "cut" (as described in section 2) would be required to make any expansion of one dimension of an element in the layout. Thus, to introduce a driver of length l and breadth b at the head of a wire, we would require a cut of width l perpendicular to the wire, and a cut of width b, parallel to the wire, both at the head of the wire, in order to accommodate the driver. Of course, in particular layouts, it may so happen that components are sparse at this region, and so the driver can be introduced as it is without any expansion of the layout. But we are looking at worst-case situations, and we will thus assume that the introduction of any element can be achieved only by making cuts of the appropriate widths. Similarly, for wide wires in fault-tolerant computing, the width of a wire is increased to w by making a cut of width w parallel to the wire.

Our general construction is as follows:

0) Lay out the circuit (or assume that the circuit layout is given).

1) Number the spatial elements that are to be varied (usually this is a subset of the lengths and widths of the lines, or the dimensions of processing elements or drivers, but could conceivably include other objects) as $v_1, v_2, \cdots, v_r$ in some arbitrary order.

2) For each spatial element v_i, identify the other elements v_j that are made to expand by its presence. For each such element, set $\delta_{ji}=k$, where v_j is made to expand k times by the presence of v_i.

3) The following system of equations defines the final values of the v_i's:

$$v_i = f_i\left(l_i + \sum_{j=1}^{r} \delta_{ij} v_j\right), \quad i = 1, 2, \cdots, r,$$

where l_i is a constant corresponding to the initial value of some linear dimension in the layout, and f_i determines the functional dependence of v_i on the v_j's. If, for each f_i, the behavior function g_i is sublinear with breakpoint r_i, then an upper bound for the v_i's is given by R, where

$$R = \max_i r_i.$$

Examples:

1) For drivers embedded in a square, v_i is the length of a side of the driver of the ith line, and l_i is the length of the ith line, and $f_i(x)=k\, x^p$, for some constant k, and for some $p<1$.

2) For fault-tolerant computing, w_i is the width of the ith line, l_i is its length, and $f_i(x)=k \cdot \ln x$, for some constant k.

8. Conclusion

In this paper we have analyzed the effect of introducing drivers to speed up signals along many long wires in a VLSI layout. We have shown that, under all but the most naive of designs for a driver (i.e., the case when drivers have constant width, and occupy area proportional to the length of the wire they drive), these drivers can be introduced with only a polynomial increase in area. With the additional assumption that drivers can be embedded in a square region, we have shown that the area at most squares by their introduction. All results have matching upper- and lower-bounds. We have also shown an area-delay trade-off in the design of drivers, and we have generalized the upper-bound proof technique. Some open problems that remain are:

1) Given a VLSI circuit, is it possible to design a layout for it so that the presence of many long wires is "minimized" according to a suitably defined criterion? Some work has been done on minimizing the length of the longest wire in a layout for certain classes of graphs. [PRS81] and [RS81] have derived bounds for trees and [BL82] have extended the result to classes of graphs with separator $n^{1/2-\varepsilon}, \varepsilon>0$ or $\sqrt{n}\log^k n, k\geq 0$. Leiserson's layout technique [L80] minimizes the maximum edgelength for classes of graphs with separator n^r, $r>1/2$. Some lower-bound results are presented in Leighton [L81]. However, our results on drivers indicate that it is not the presence of a single long wire that causes the worst-case increase in area, but rather the presence of *many* long wires in a configuration that requires many cuts. Even heuristics to prevent such undesirable configurations of long lines should be very useful.

2) Is it possible to incorporate the presence of drivers in our layout model, for example, by using nodes of variable size, and can we obtain area-efficient layouts under this model? Using such a model, we can re-examine the upper-bounds that have been obtained for the layout area of many common circuits. In particular, upper-bounds for $AT^\alpha, \alpha>0$, *should* be re-examined under the context of area-expansion caused by drivers of long wires.

3) The numerical solution of equation 2: In section 6, we used the Brouwer fixed-point theorem to prove results on the existence and worst-case behavior of the solution of equation 2 under various assumptions. However, the actual solution of the system of equations for particular layouts still needs to be investigated.

4) Modeling poly lines in MOS technology: We have modeled the wire as a purely capacitive load. However, poly lines have rather high resistance, and may be better modeled as distributed RC-networks. At present, the best that can be done for such structures is to use a driver of constant size after each fixed interval of the wire and this reduces the delay from $\Theta(l^2)$ to $\Theta(l)$, where l is the length of the poly line. It will be very useful to find a better way to speed up signals along such wires.

Acknowledgements I am greatly indebted to Professor Richard Lipton for suggesting the problem, for useful discussions and comments, and in particular, for drawing my attention to the Brouwer fixed-point theorem. I also wish to acknowledge help from Andrea LaPaugh in the field of VLSI design, and from Bradley Dickinson in the theory of M-matrices. I would like to thank all of the above, and Karl Leiberherr and Avi Wigderson for comments that led to a better organization of the paper.

References

[B64] R. G. Bartle, *The Elements of Real Analysis*, John Wiley and Sons, Inc., New York, 1964.

[BP79] A. Berman and R. J. Plemmons, *Nonnegative Matrices in the Mathematical Sciences*, Academic Press, New York, 1979.

[BL82] S. N. Bhatt and C. E. Leiserson, "Minimizing the longest edge in a VLSI layout," manuscript, Lab. for Computer Science, MIT, Cambridge, Mass.

[BPP81] G. Bilardi, M. Pracchi, and F. P. Preparata, "A critique and appraisal of VLSI models of computation," in *CMU Conference on VLSI Systems and Computations*, H. T. Kung, Bob Sproull, and Guy Steele, eds., Oct. 1981, pp. 81-88.

[BK80] R. P. Brent and H. T. Kung, "The chip complexity of binary arithmetic," in *Proc. of the 12th Annual Symposium on the Theory of Computing*, Los Angeles, April 1980, pp. 190-200.

[CM81] B. Chazelle and L. Monier, "A model of computation for VLSI with related complexity results," in *Proc. of the 13th Annual Symposium on the Theory of Computing*, May 1981, pp. 318-325.

[J75] R. C. Jaeger, "Comments on 'An optimized output stage for MOS integrated circuits'," *IEEE J. Solid State Circuits*, June 1975, pp. 185-186.

[L81] F. T. Leighton, "New lower bound techniques for VLSI," in *Proc 22nd Symp. on the Foundations of Computer Science*, IEEE Computer Society, Oct. 1981, pp. 1-12.

[L80] C. E. Leiserson, "Area efficient graph embeddings (for VLSI)," in *Proc. 21st Symp on the Foundations of Computer Science*, IEEE Computer Society, Oct. 1980, pp. 270-281.

[LL75] H. C. Lin and L. W. Linholm, "An optimized output stage for MOS integrated circuits," *IEEE J. Solid State Circuits*, April 1975, pp. 106-110.

[MC80] C. A. Mead and L. C. Conway, *Introduction to VLSI Design*, Addison-Wesley, Reading, Mass. 1980.

[MR79] C. A. Mead and M. Rem, "Cost and performance of VLSI computing structures," *IEEE J. Solid State Circuits*, April 1979, pp. 456-462.

[PRS81] M. S. Paterson, W. L. Ruzzo, and L. Snyder, "Bounds on minimax edge length for complete binary trees," in *Proc. of the 13th Annual Symposium on the Theory of Computing*, May 1981, pp. 293-299.

[RS81] W. L. Ruzzo and L. Snyder, "Minimum edge length planar embedding of trees," in *CMU Conference on VLSI Systems and Computations*, H. T. Kung, Bob Sproull, and Guy Steele, eds., Oct. 1981, pp. 119-123.

[T80] C. D. Thompson, *A Complexity Theory for VLSI*, Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, 1980.

[T81] C. D. Thompson, "The VLSI complexity of sorting," in *CMU Conference on VLSI Systems and Computations*, H. T. Kung, Bob Sproull, and Guy Steele, eds., Oct. 1981, pp. 108-116.

[vN56] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. E. Shannon and J. McCarthy, eds., *Annals of Mathematical Studies* No. 34, Princeton University Press, Princeton, NJ, 1956.

[V80] J. Vuillemin, "A combinatorial limit to the computing power of VLSI circuits," in *Proc 21st Symp on the Foundations of Computer Science*, IEEE Computer Society, Oct. 1980, pp. 294-300.

END

DATE
FILMED

1-84

DTIC